ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

GENERACIÓN AUTOMÁTICA DE MECÁNICAS DE JUEGO EN UN JUEGO RTS

PROCEDURAL GENERATION OF GAME MECHANICS IN AN RTS GAME

Realizado por
Alejandro Ruiz Moyano

Tutorizado por
Antonio José Fernández Leiva
Mariela Nogueira Collazo

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE 2015

Fecha defensa:
El Secretario del Tribunal

Resumen: Cada vez se extiende más el uso de técnicas de generación automática de contenidos dentro del ámbito de la creación de videojuegos. Estos contenidos abarcan elementos tan variados como pueden ser personajes, música, edificios, bosques o climatología, pero no incluyen a las propias reglas del juego, debido a la enorme dificultad que ello conlleva y al hecho de que la variación de las reglas supone, implícitamente, cambiar el juego.

Es por esto que los objetivos del presente trabajo son dos principalmente: el desarrollo de una aplicación de escritorio para editar las reglas de un juego de estrategia en tiempo real y realizar una experimentación preliminar para evolucionar las reglas del propio juego, a través del empleo de técnicas de inteligencia artificial avanzada (concretamente, mediante métodos bio-inspirados dentro de la computación evolutiva).

Para ello será necesario realizar, inicialmente, un estudio de la lógica del juego para parametrizar sus reglas y permitir al usuario modificar las mismas de forma simple. Posteriormente, se diseñarán e implementarán diversos algoritmos genéticos que permitan generar reglas de juego y probar la eficacia de las mismas.

De esta forma, se pretende que este trabajo conforme los primeros pasos hacia la generación automática de nuevos juegos.

Palabras claves: generación automática, videojuegos, reglas lógicas, PCG, *Eryna*, algoritmos genéticos, inteligencia artificial, computación evolutiva.

Abstract: The use of procedural content generation techniques are spreading out increasingly in the videogames creation field. This content encompasses varied elements as can be characters, music, buildings, forests or weather conditions, but it doesn't include the rules of the game itself, because of the huge difficulty that it entails and because of the fact that the variation of the rules implicitly means changing the game.

This is why the objectives of the present project are mainly two: the development of a desktop application to edit the rules of a game and making a preliminary experimentation to evolve the rules of the game itself, by the employment of advanced artificial intelligence techniques (in particular, using bio-inspired methods of the evolutionary computation).

To achieve this, it will be necessary to perform, initially, a study of the game's logic to parametrize its rules and allow the user to modify them in a simple way. Also, it will be necessary to design and to implement diverse genetic algorithms that can generate game's rules and to test their efficiency.

In this way, it's expected that this project defines the first steps towards the procedural generation of new games.

Keywords: procedural generation, videogames, logic rules, PCG, *Eryna*, genetic algorithms, artificial intelligence, evolutionary computation.

ÍNDICE DE CONTENIDO

Introducción

El videojuego es el producto cultural que mayor proyección de crecimiento tiene en el mundo. Es un sector que lleva varios años produciendo más actividad económica que el resto de sectores culturales y que sigue creciendo exponencialmente debido a su excelente aceptación [DEV, 2015]. Según diferentes consultoras, el mercado de los videojuegos seguirá creciendo sostenidamente a lo largo de los próximos años. Por ejemplo, la consultora PWC estima que en 2019 el mercado global de videojuegos ascienda a 93.180 millones de dólares [PWC, 2015].

A la luz de estos datos es normal que los videojuegos hoy en día sean percibidos como algo muy cotidiano. Personas de todas las edades han jugado alguna vez a alguno y las ventas de consolas se cuentan por decenas de millones alrededor del mundo. Es raro no ver en alguna casa alguna consola (ya sea portátil o de sobremesa) o algún ordenador en el que se tengan videojuegos. Pero el desarrollo de videojuegos va más allá de los fines lúdicos, tal es el caso de los llamados juegos serios. Incluso ahora se *gamifican* ciertas actividades como por ejemplo salir a correr, programar en un nuevo lenguaje o aprender un nuevo idioma.

Resulta sorprendente pues mirar hacia atrás unos 50 años y ver la evolución que han tenido los videojuegos desde aquel entonces: empezando por el ya “prehistórico” *Pong*, pasando por las máquinas recreativas tan de moda en las décadas de los 80 y 90, y finalizando en los juegos de gráficos ultra realistas que proliferan en la actualidad.

Debido a esta evolución, la cantidad de personas y el tiempo requerido necesarios para desarrollar videojuegos han ido incrementando cada vez más, hasta el punto de que, en la actualidad, son necesarias cientos de personas y años de desarrollo para los lanzamientos más prestigiosos [Togelius et al., 2015]. Un equipo de desarrollo de esta clase de videojuegos incluye programadores (de diferentes tipos), guionistas, músicos, artistas conceptuales, modeladores, diseñadores de interfaces y un largo etcétera de profesionales de diferentes campos que se pueden involucrar en la creación de un mismo videojuego.

Dentro de los videojuegos hay muchas categorías o géneros: acción, aventuras, deportivos, FPS (*First Person Shooter*, por sus siglas en inglés), rol (pueden ser online masivos u offline), RTS (*Real-Time Strategy*, por sus siglas en inglés), terror, plataformas, MOBA (*Multiplayer Online Battle Arena*, por sus siglas en inglés), etc. De entre estos géneros cabe destacar dos: los MOBA, por su popularidad en la actualidad y porque se les atribuye la mayor parte del éxito de los conocidos como *e-sports* o deportes electrónicos; y los RTS en especial por ser objeto del presente proyecto.

Los juegos de corte RTS son muy interesantes porque tienen una lógica bastante compleja debido a que los jugadores deben ser capaces de gestionar diversos recursos para construir unidades, bases, y diseñar diferentes estrategias para su bando. En la actualidad, este tipo de juegos tiene una gran aceptación entre los jugadores, de forma que incluso en Corea del Sur se retransmiten por televisión partidas competitivas de *Starcraft II* de la desarrolladora Blizzard [Blizzard, 2015]. También merece una mención la saga de videojuegos RTS *Age of Empires* [Microsoft, 2015], que gozó de gran popularidad a principios de los años 2000 y que basa su apuesta jugable en el género más clásico de los RTS con ambientación histórica.

Si se analizan las tendencias modernas del desarrollo de videojuegos se destaca una técnica que ha ido surgiendo en los últimos tiempos: la Generación de Contenido por Procedimientos [Togelius et al., 2011]. PCG (*Procedural Content Generation*, por sus siglas en inglés) es una técnica capaz de agilizar y facilitar el trabajo de los desarrolladores y que consiste en la generación automática, mediante diversos algoritmos (en lugar de manualmente), de contenido para videojuegos.

Este contenido generado algorítmicamente abarca desde los mapas y niveles completos del juego a características de las armas y de cualquier otro ítem e, incluso, ciertas reglas del propio juego [Gutierrez et al., 2014].

La historia de PCG se remonta hasta 1980, donde un videojuego llamado *Rogue* basaba su mapeado en una serie de mazmorras que estaban compuestas por diversas habitaciones conectadas entre sí con pasillos. Este juego generaba aleatoriamente, mediante técnicas PCG, las habitaciones, los pasillos, el posicionamiento de los objetos e, incluso, laberintos en los niveles más avanzados. Más adelante, en 1995, se publicó *Diablo*, un videojuego de acción y RPG (el primero de una saga que hoy en día sigue bastante activa) en el que también se hacía uso de PCG para aleatorizar el diseño de sus mazmorras y la generación de objetos y enemigos [Togelius et al., 2015]. Finalmente, en la actualidad tenemos el caso de *Minecraft*, un juego de gran fama en el que sus mapeados son generados de forma aleatoria [Mojang, 2015].

Existen varios ejemplos de técnicas de PCG que han sido usadas en diversos juegos, la más famosa y extendida es la generación de niveles aleatorios en tiempo de ejecución. Videojuegos como los mencionados anteriormente hacen uso de esta técnica para, mientras el juego está en ejecución, se generen los escenarios por los que pasará el jugador. También los diseñadores de nivel pueden apoyarse en PCG durante el diseño de los mapas para generar automáticamente el terreno de los mismos y luego llenarlos de diferentes objetos a mano reduciendo el tiempo de desarrollo. Otra alternativa pasa por aplicar PCG a los elementos del juego como por ejemplo enemigos o ítems para generar, de forma aleatoria, su posición dentro del mapeado o, incluso, su comportamiento.

El objetivo de este proyecto es la aplicación de técnicas PCG para generar de forma automática las reglas lógicas en un juego de tipo RTS y poder, de este modo, obtener distintas mecánicas que darán lugar a otras versiones del propio juego. Para ello se usará *Eryna* [Nogueira et al., 2014], que es un videojuego desarrollado en la Universidad de Málaga y que sirve de herramienta de apoyo a los investigadores de IA (Inteligencia Artificial) poniendo a su disposición módulos relacionados con los videojuegos adaptativos, videojuegos afectivos y aplicaciones de PCG.

En la literatura científica consultada se encuentran muy pocos resultados en lo que respecta a la generación automática de reglas en videojuegos. De hecho, en videojuegos de estrategia nunca antes se ha hecho, de ahí que el aporte de este proyecto es relevante en este campo y constituye un paso inicial hacia la generación automática de un juego completo. Siendo esto último una línea de investigación abierta por la cual se está apostando en la comunidad científica y es, además, una tarea compleja pues supone un enfoque de diseño de videojuegos diferente, que permita al motor del juego ser parametrizable, y generar y a la vez adaptarse a estados lógicos no previstos inicialmente.

Para dar cumplimiento al objetivo de esta investigación se han definido varias tareas que incluyen: la parametrización de la lógica inicial del juego que permita una configuración simple al alcance del usuario, el desarrollo de una aplicación de escritorio que permita editar las reglas del juego a gusto del usuario y el diseño y aplicación de algoritmos bio-inspirados que permitan generar automáticamente diferentes versiones del juego variando sus reglas.

La estructura de esta memoria es la que sigue. En el Capítulo 1 se abordará la Generación de Contenidos por Procedimientos y sus técnicas de aplicación más destacadas. Luego, en el Capítulo 2, se presentará el proyecto en el que se enmarca esta investigación. A continuación, en el Capítulo 3 se explicará en detalle la aplicación de escritorio desarrollada. Seguidamente, en el Capítulo 4, se profundizará en los algoritmos genéticos y sus principales parámetros. Finalmente, en el Capítulo 5 se presentarán y analizarán los resultados obtenidos en esta investigación.

Capítulo 1

Generación de Contenido por Procedimientos

Este capítulo tratará sobre la Generación de Contenido por Procedimientos (PCG de ahora en adelante, por sus siglas en inglés) y de algunas de las diferentes técnicas disponibles para aplicarla en videojuegos.

Se definirán los conceptos básicos necesarios, se repasarán las principales ventajas de su uso y se detallarán las técnicas de aplicación. También se hará un breve viaje a través de la historia de PCG, enumerando algunos de los juegos más emblemáticos que hacen uso de ella.

1.1 Introducción a PCG

Togelius et al. definen PCG como la creación algorítmica de contenido de juego con participación limitada o indirecta del usuario [Togelius et al., 2015]. Esto quiere decir que los diseñadores de juego pueden apoyarse en PCG para desarrollar sus tareas o, simplemente, dejarla que cree contenidos por sí misma.

El tipo de contenido que se puede generar con PCG abarca prácticamente cualquier elemento de un videojuego: mapas, objetos, personajes, armas, misiones, música e, incluso, reglas de juego. El hecho de que PCG pueda genera reglas del propio juego es la principal motivación detrás del presente proyecto y, por ello, nos centraremos en ellas más adelante.

Existen varias ventajas a la hora de usar técnicas PCG para desarrollar videojuegos. La primera (y más obvia) razón es la reducción drástica del tiempo y del coste de desarrollo de un videojuego: se pueden sustituir varios artistas y diseñadores por este tipo de algoritmos para reducir estos costes o bien, estos mismos artistas y diseñadores pueden usar PCG como apoyo a su trabajo. Otro punto fuerte de usar PCG es que puede ayudar a los desarrolladores a aumentar su creatividad: esto es debido a que un algoritmo puede dar como resultado un contenido radicalmente distinto a lo que podría dar un ser humano determinado y ser igualmente válido. Otra de las razones podría ser la generación de juegos “infinitos”: si un juego usa estas técnicas para generar su contenido a la misma velocidad a la que es consumido por los jugadores, éste no tendría fin. Esto es altamente deseable en juegos de perfil online como los del género MMORPG (*Massive Multiplayer Online Role Playing Game*, por sus siglas en inglés) ya que, en muchas ocasiones, una gran cantidad de jugadores de este tipo de juegos han superado la mayoría del contenido de los mismos mucho antes de que los desarrolladores lanzasen alguna expansión.

1.2 Historia de PCG

Resulta curioso ver que las técnicas de PCG ha formado parte de los videojuegos casi desde el principio de los mismos [Hendrikx et al., 2013]. Videojuegos tan lejanos en el tiempo como los ya mencionados *Rogue* (ver Figura 1.1) o *Diablo* (ver Figura 1.2) supusieron el despegue del uso de este tipo de técnicas de generación de contenido en videojuegos, una tendencia que ha ido ganando cada vez más adeptos en los últimos años.

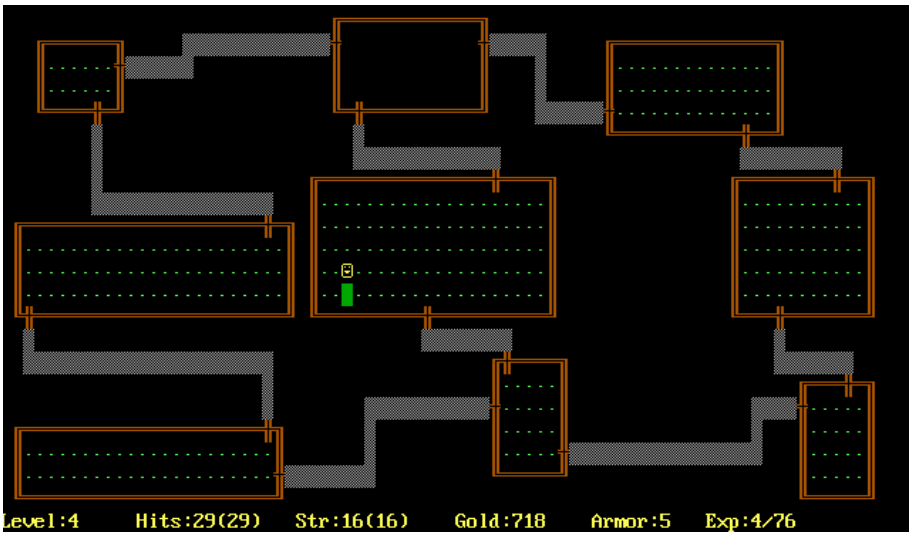

Figura 1.1. Ejemplo de mazmorra de *Rogue*

Figura 1.2. Ejemplo de mazmorra de *Diablo*

Como ejemplos de videojuegos más recientes que hacen uso de técnicas PCG podemos destacar otros dos: *Spore* y *Minecraft*.

Spore (ver Figura 1.3) es un videojuego publicado en 2008 de corte RTS y de simulación de vida que permite al jugador controlar el desarrollo de una especie desde sus comienzos como un organismo microscópico, pasando por la evolución de inteligencia, hasta llegar a la exploración espacial [EA, 2015].

Lo más interesante de *Spore* es que hace un uso intensivo de PCG, ya que el jugador diseña la fisionomía de la especie y ésta es animada mediante técnicas de animación por procedimientos; es decir, PCG aplicada a la animación como forma de contenido. Además, la galaxia en la que se desarrolla el juego también es generada usando procedimientos [Togelius et al., 2015].

Minecraft (ver Figura 1.4), por su parte, es un videojuego perteneciente al género *sandbox* que fue publicado en 2011 y que permite a los jugadores crear construcciones usando cubos de texturas en un mundo 3D generado aleatoriamente con PCG [Mojang, 2015].

Figura 1.3. Creador de criaturas de *Spore*

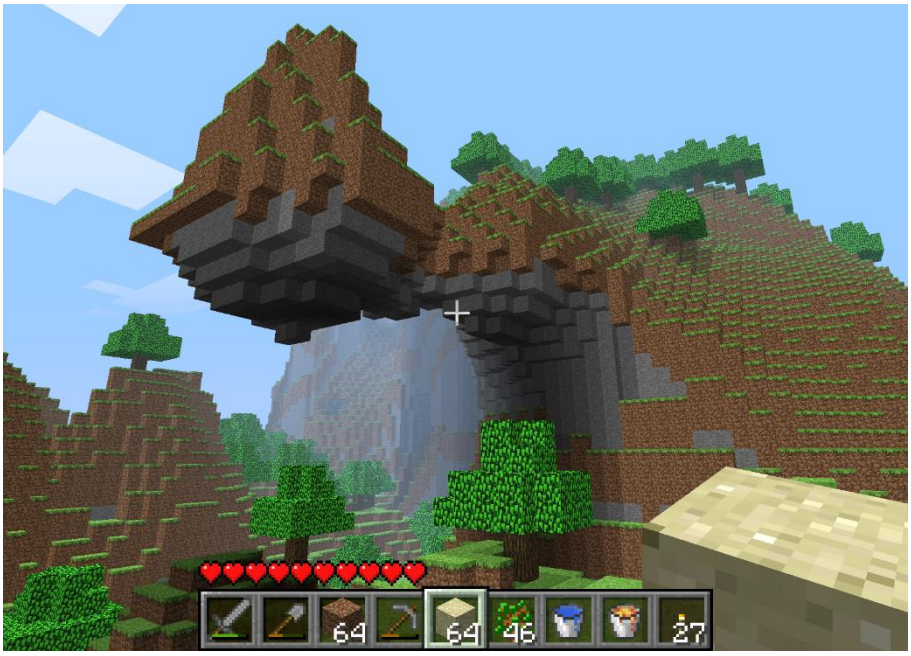

Figura 1.4. Escenario de *Minecraft*

Hoy en día, la popularidad de PCG entre la comunidad científica es bastante elevada, tal y como demuestran la gran cantidad de publicaciones que se han realizado al respecto. De hecho, en 2010 se celebró la primera competición de generación de contenido por procedimientos del mundo dentro del marco de la *2010 Mario AI Championship* [Shaker et al., 2011]. En esta competición los participantes tenían que generar, mediante PCG, niveles para una versión del juego *Super Mario Bros*: esto incluye generar el terreno, las tuberías, los enemigos, los bloques y las monedas para cada mapa.

1.3 Técnicas de aplicación de PCG

Existen muchos tipos de técnicas de PCG, una de las principales categorías de las mismas es la de generación de mapas (que ya fue introducida con anterioridad). Dentro de esta categoría hay muchas técnicas disponibles, aunque todas tienen un mismo objetivo en común: generar los elementos necesarios para crear un mapa para un videojuego.

Como ejemplos de estas técnicas se pueden nombrar los diferentes algoritmos que existen para generar edificios, bosques, terreno (montañas, mares, etc.), vegetación y elementos dinámicos como pueden ser fuego, agua, condiciones meteorológicas y vida artificial.

También podemos clasificar en la categoría de generación de personajes todas aquellas técnicas relacionadas con los mismos: generar animaciones, modelos, árboles de conversación, ropa, nombres, objetos, etc.

Estos ejemplos no son las únicas técnicas de aplicación de PCG que hay disponibles actualmente, de hecho hay bastantes más: para generación de música, de arte, de puzles, incluso se puede programar la dificultad del juego de forma que se vaya adaptando a la capacidad de cada jugador de forma automática [PCGWiki, 2015].

Cabe destacar que las técnicas de generación de elementos naturales se clasifican en dos vertientes: la teleológica y la ontogénica. La categoría teleológica engloba todas aquellas técnicas que, para obtener el resultado requerido, hacen simulaciones de los procesos físicos que se llevarían a cabo en la naturaleza para ello. Sin embargo, las técnicas ontogénicas se basan en replicar el resultado final de esos procesos físicos [West, 2008].

A modo de ejemplo sencillo, una técnica teleológica para generar montañas realizaría primero la simulación del choque de las placas tectónicas bajo el suelo plano originando las montañas. Por otro lado, una técnica ontogénica simplemente crearía esas montañas emulando las de la naturaleza de forma aleatoria.

1.4 Futuro del PCG

En esta sección se presentan algunas de las líneas de investigación en las que pueden ir encaminadas las técnicas PCG para alcanzar una serie de metas que, hoy por hoy, son inalcanzables. Se presentarán estas metas y las dificultades principales que hay que superar para alcanzarlas.

1.4.1 PCG multi-nivel y multi-contenido

El objetivo final de esta investigación sería poder implementar mundos virtuales con sólo un *click* de ratón. Se podría generar terrenos, vegetación, ciudades, personajes, misiones, objetos, vehículos, caminos, textos, diálogos, texturas, etc. Un mundo como el de *Skyrim* (ver Figura 1.5) de la desarrolladora Bethesda Game Studios [Bethesda, 2015b], con una vasta extensión jugable y con infinidad de detalles, personajes, texturas y demás elementos sería posible crearlo con sólo pulsar un botón. Además, se podrían crear mundos de diferentes estilos: ciencia ficción, post-apocalípticos, medievales…

Por supuesto, estos mundos estarían basados en las especificaciones del motor gráfico sobre el que se crease el mundo, adaptándose a las acciones posibles, interacciones, sistemas de puntuación y demás reglas que definen al juego que estén definidas en el motor.

Figura 1.5. Paisaje en The Elder Scrolls V: Skyrim

El principal problema para lograr esto es que, en general, los contenidos generados automáticamente son bastante genéricos. Por ejemplo, las mazmorras generadas en la saga de juegos *Diablo* (mencionada con anterioridad) son diferentes entre sí pero muchas de ellas dan la sensación de ser muy parecidas a las demás. Es por ello, que el reto de esta línea de investigación consiste en crear generadores que creen contenido creativo, significativo y original.

1.4.2 Diseño de juego basado en PCG

Esta vertiente de PCG se basa en la elaboración de juegos en los que PCG jugase un papel vital de forma que, sin PCG los juegos no podrían ser concebidos siquiera. Esto daría lugar a un nuevo género de videojuegos que se basaría en la exploración de contenidos realmente infinitos.

Para llevar esta meta acabo serían necesarias varias innovaciones tanto en las metodologías de diseño de juegos como en las propias técnicas PCG: el diseño de juego debería tener en cuenta que los parámetros para el generador de contenido tiene que estar altamente basado en las acciones de los jugadores y el algoritmo PCG debe ser mucho más fiable y controlable que los disponibles actualmente.

Hay algunos videojuegos que han logrado avances en este sentido y, por ejemplo, en *Infinite Tower Defense* la misión del algoritmo PCG es crear nuevos niveles y enemigos que contrarresten la estrategia actual del jugador de modo que éste se vea forzado a explorar nuevas estrategias [Avery et al., 2011].

1.4.3 Generación de juegos completos

Por último, la meta de esta línea es la de la creación de juegos completos por parte de un sistema PCG; es decir, no sólo implica crear el mundo al completo, si no crear además las reglas del juego y el motor gráfico.

Esto implicaría automatizar los aspectos más centrales del diseño de juego, incluyendo la estimación de cómo un humano puede experimentar la interacción con un sistema completo de reglas. Por tanto, uno de los principales escollos para que la generación de juegos completos sea posible (además de los problemas intrínsecos del uso de PCG multi-nivel y multi-contenido) es la generación automática de reglas para dichos juegos. Y de aquí nace la importancia del presente proyecto, debido a que se han producido muy pocos avances en este aspecto.

En este proyecto en concreto, se presentan las dificultades de generar reglas aleatorias para un juego ya existente; es decir, se modifica totalmente el juego del que se parte y, además, hay que evaluar si esta modificación es mejor o peor que el original.

Capítulo 2

Proyecto *Eryna*

Este capítulo versará sobre *Eryna* (proyecto desarrollado en el departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga) [Nogueira et al., 2014] y se expondrán sus motivaciones, su estructura y su relación con la propuesta que se realiza en esta investigación.

2.1 Motivaciones detrás de *Eryna*

El juego, en todas sus modalidades, persigue un objetivo tan complejo como es el de proporcionar entretenimiento o diversión al jugador humano, incluso los videojuegos serios independientemente de sus fines educativos, instructivos, terapéuticos, etc. tienen que proporcionar su dosis de entretenimiento o, de lo contrario, serían rechazados por los usuarios. Y de esta sencilla razón surgen las cuestiones más interesantes del tema: ¿Cómo medir la satisfacción del jugador? ¿Cómo desarrollar un videojuego que complazca a muchas personas que tienen gustos, culturas, habilidades, y expectativas diferentes? No son pocos los que se han implicado en responder a estos interrogantes, psicólogos, matemáticos, informáticos, y muchos emprendedores fieles de diferentes culturas, subculturas y profesiones, han aportado respuestas, muchas de ellas las podemos encontrar si revisamos los resultados de la actividad científico-investigadora sobre videojuegos en los últimos años. Las propuestas más relevantes apuestan por explotar técnicas de Inteligencia Computacional (IC) que doten a los oponentes virtuales de una Inteligencia Artificial personalizada a las necesidades y características de cada jugador, que hagan de cada partida una experiencia única y retadora. En sintonía con este objetivo existen tres tendencias que han ganado protagonismo en el auge científico-investigativo de la Inteligencia Artificial aplicada a videojuegos. Una de ellas es la ya mencionada (en el capítulo anterior) PCG y las otras dos se abordarán en las siguientes secciones.

2.1.1 Videojuegos auto-adaptativos

Esta línea de investigación se basa en la posibilidad de que los videojuegos puedan adaptarse por sí mismos a las preferencias de cada jugador. Para lograr este objetivo todos los elementos que componen un videojuego deberían ser creados de forma genérica para que las partidas sean flexibles y adaptables. Esto quiere decir que un mismo videojuego podría resultar en una experiencia completamente diferente en función del jugador que lo juegue ya que, idealmente, el juego dispondría de infinitas combinaciones de sus elementos ajustables.

Es fundamental, pues, que haya algún mecanismo de retroalimentación del estado del jugador en cada momento para configurar los ajustes de la partida en consonancia al mismo. El principal problema reside en que estos mecanismos son altamente complejos debido a la cantidad de factores que influyen en el

estado de ánimo de una persona y a que cada una tiene unas características diferentes.

En la actualidad, se realiza el *IEEE Task Force on Player Satisfaction Modeling* [IEEE, 2015] para incentivar la creación de propuestas de modelado y optimización de la medida de satisfacción de los jugadores. En principio, hace falta obtener los modelos que permitan identificar al jugador y luego usar técnicas psicológicas o dispositivos biométricos para obtener la retroalimentación del jugador mientras juega y, en este sentido, se haya la investigación de esta tendencia.

Cabe destacar que algunos videojuegos ya usan sistemas de dificultad adaptativa en sus partidas en función de diferentes factores. *Mario Kart Wii* (ver Figura 2.1) y todos los demás juegos de la serie *Mario Kart* de la desarrolladora Nintendo [Nintendo, 2015], ajusta los ítems que los jugadores pueden obtener en cada carrera en función de la posición en la que vayan: los jugadores más atrasados tienen unas probabilidades bastante altas de que les toque los ítems más poderosos, mientras que los jugadores más avanzados tienen muy pocas probabilidades de ello. Por otro lado, *Fallout 3* (ver Figura 2.2) es un videojuego de acción RPG publicado por Bethesda Game Studios en 2008 [Bethesda, 2015a] que incrementaba las estadísticas y mejoraba las armas de los enemigos a medida que el jugador iba subiendo de nivel, incluso se reemplazaban algunos enemigos por otros más poderosos. Finalmente está el caso de *Grand Theft Auto 5* (ver Figura 2.3), videojuego de corte *sandbox* publicado por Rockstar Games en 2013 [Rockstar, 2015], que dispone de un sistema de ayuda en sus misiones: si un jugador falla una misión en algún punto de la misma 3 veces seguidas, se le ofrece la opción de saltar automáticamente al siguiente *checkpoint* de la misión.

Figura 2.1. Carrera en *Mario Kart Wii*

Figura 2.2. Enemigos en *Fallout 3*

Figura 2.3. Omitir sección fallida en *GTA 5*

2.1.2 Videojuegos afectivos

Esta tendencia guarda una estrecha relación con la anterior ya que ambas se centran en las emociones humanas aunque con un enfoque diferente.

En 1995 Rosalind Picard introdujo el término de Computación Afectiva (CA) y que definió como “el cómputo que relaciona, surge o influye en las emociones” [Picard, 1995]. En el campo de los videojuegos, la Computación Afectiva se introdujo al principio mediante la narrativa de la historia: por ejemplo, se pueden estimular las emociones de los jugadores creando situaciones o personajes con los que se pueda relacionar. Videojuegos como *Fahrenheit* (ver Figura 2.4) o *Heavy Rain* (ver Figura 2.5), publicados en 2005 y 2010 respectivamente por la desarrolladora Quantic Dream ([Quantic, 2015a] y [Quantic, 2015b]), dejan a un lado la jugabilidad más clásica de los géneros de acción para dar paso a un sistema de narración interactiva con QTE (*Quick Time Events*, por sus siglas en inglés), lo cual provoca que el jugador se vea muy inmerso en sus respectivas historias.

Sin embargo, esta rama de investigación va más allá de provocar reacciones emocionales en los jugadores y propone que también los personajes

o *bots* de los videojuegos sean capaces de reaccionar a las señales emocionales, tanto del jugador como de otros *bots*. Por ello, al igual que en el caso de los videojuegos auto-adaptativos, es necesario algún mecanismo de retroalimentación del jugador para que el juego, en función de las señales emocionales recibidas, ajuste las respuestas emocionales necesarias.

Cabe destacar una empresa que se caracteriza por hacer especial hincapié en la IA de sus videojuegos: Lionhead Studios. Con uno de sus principales juegos, *Fable*, tratan de generar emociones realistas en sus *bots* para causar reacciones en los jugadores [Lionhead, 2015].

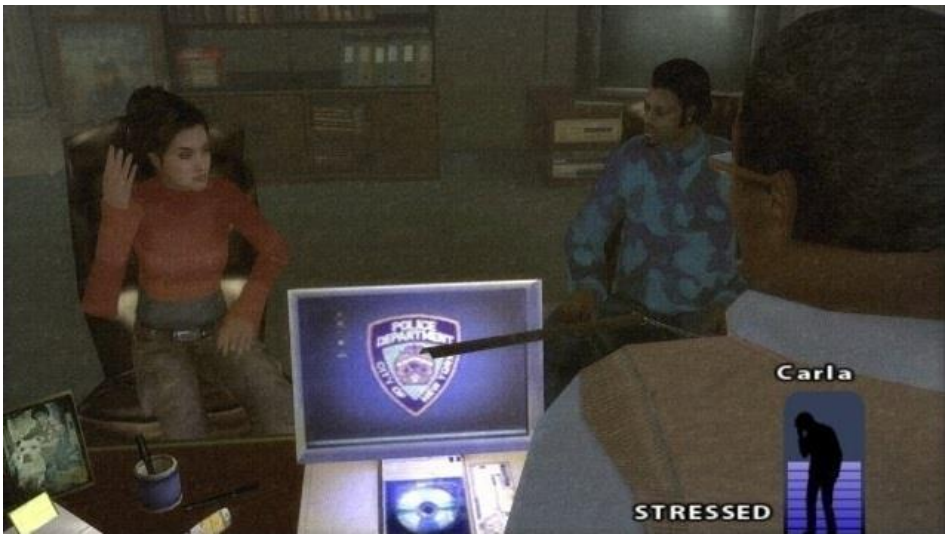

Figura 2.4. Estado mental de un personaje de *Fahrenheit*

Figura 2.5. Posibles interacciones en *Heavy Rain*

2.2 Objetivos y descripción de *Eryna*

Varias son las plataformas de prueba que han sido implementadas para apoyar el trabajo de los investigadores en las nuevas tendencias de los videojuegos, la mayoría están enfocadas a la evaluación de mecanismos de IA para optimizar el comportamiento de los jugadores virtuales (*bots*). Por ejemplo está *ORTS*, el cual es un juego de estrategia de tiempo real que proporciona un cliente que permite la aplicación de técnicas de aprendizaje automatizado a su motor de juego [ORTS, 2015]. También se han usado otros juegos que han sido objeto de competiciones para fomentar la creación de técnicas de IA aplicada, como es el caso del *Google AI Challenge Contest* [AI, 2015]*,* cuyos juegos han sido objeto de muchas investigaciones sobre IA.

Eryna es otra herramienta de apoyo a la investigación similar a las demás que existen en la actualidad, con la distinción de que ha sido diseñada especialmente para ser integrada en algoritmos que exploren técnicas avanzadas de IA para videojuegos y ofrece a los investigadores facilidades como: la fácil reproducción de sus escenarios de juegos y de sus *bots* para que puedan ser objeto de experimentos científicos; la exportación de *logs* de las partidas en un formato extendido y también configurable; la posibilidad de ejecutar partidas multijugador para evaluar comportamientos de IAs en entornos cooperativos; y la fácil parametrización de las reglas lógicas que determinan las partidas y crean mecánicas de juegos distintas, siendo este punto el objetivo de la presente investigación.

2.2.1 Arquitectura del sistema

A continuación se presenta la Arquitectura del Sistema *Eryna*. En la Figura 2.6 se puede observar una vista lógica de *Eryna* como Plataforma Interactiva de Prueba y Optimización donde los componentes principales son:

- ***Game_Engine*:** implementa una versión no gráfica de un juego RTS multijugador, donde cada jugador (virtual o humano) compite con los demás para ser el único ganador. La mecánica del juego es la de un clásico juego en un entorno espacial en la que los usuarios conquistan mundos; pero en este caso la lógica del juego se complica, pues habrá que manejar un conjunto de recursos como sucede típicamente en los juegos de estrategia. Cada jugador contará con un grupo de unidades y recursos que deberá ir repartiendo por diferentes mundos para así ir aumentando sus conquistas y su poder.
- ***AI_Engine:*** es el módulo que permite definir IAs (es decir, *bots*) para jugar en *Eryna*. Ofrece una serie de clases de tipo interfaz que se deben redefinir para implementar el comportamiento específico del *bot* que se desee programar.
- ***Results_Manager*:** un módulo que se encarga de gestionar los datos relevantes de las partidas ejecutadas y permite que el usuario configure los datos que quiere recopilar.
- ***Rules_Editor*:** se refiere a una de las soluciones que aporta el presente proyecto (detallada en el Capítulo 3), la cual es una aplicación de

escritorio que permitirá editar y definir de forma fácil las reglas lógicas de las partidas para que el usuario pueda ajustar las mecánicas de juego que desee que se generen en sus experimentos.

- ***Rules_Generator*:** es parte también de la solución que aporta esta memoria (se detalla en el Capítulo 5), se trata de un módulo que basado en un algoritmo bioinspirado permite la generación automática de las reglas que influyen en la lógica del juego.

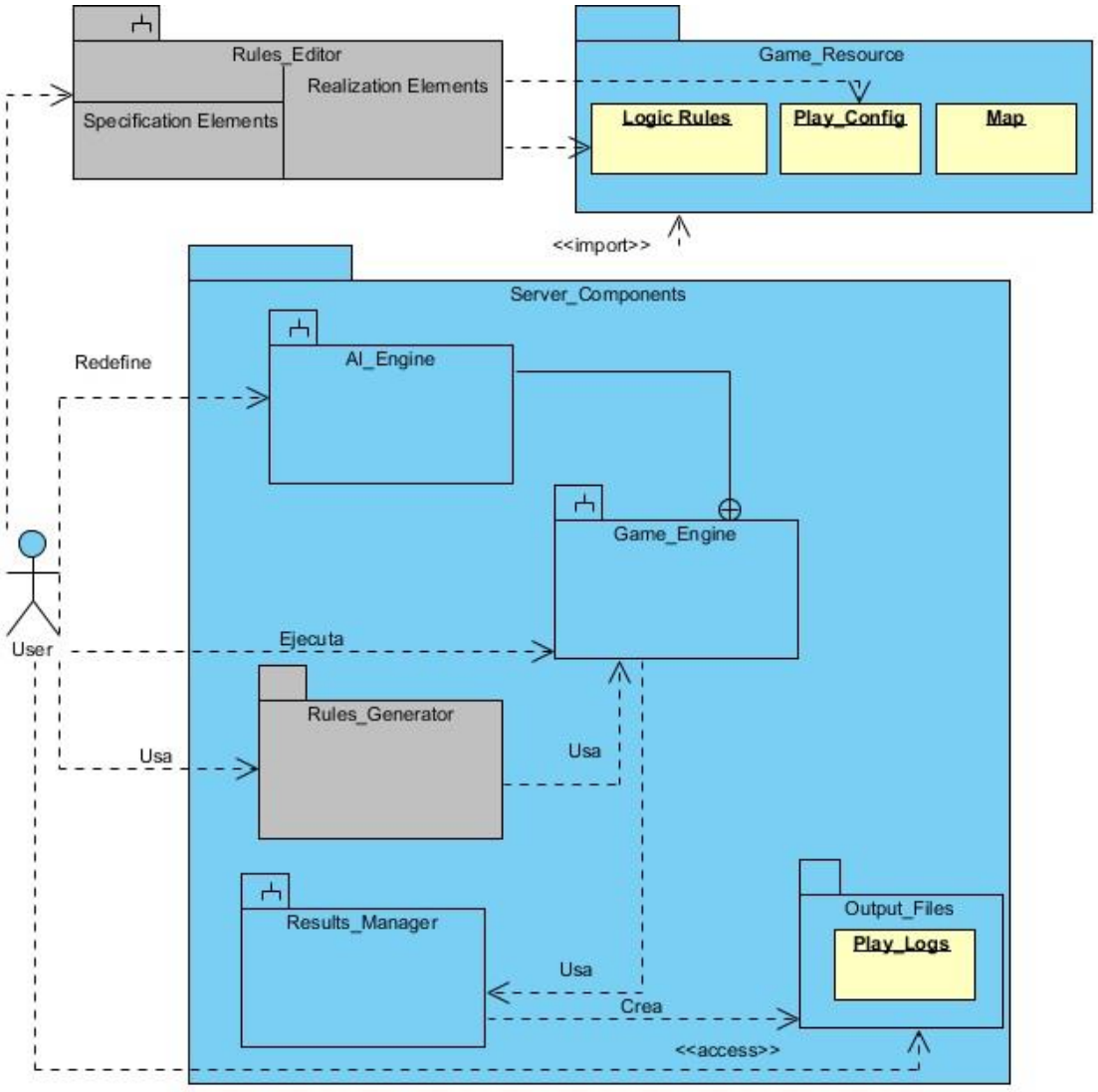

Figura 2.6. Plataforma Interactiva de Prueba y Optimización

2.2.2 Descripción del escenario de juego

El escenario de juego está determinado por el mapa, los jugadores (o *bots*) y las reglas configurables que condicionan la mecánica de la partida. Un mapa está compuesto por varios mundos/planetas y de cada uno se conoce: la posición en la que se encuentra; la pertenencia, que indica si es un planeta conquistado por un jugador; y los recursos que tiene. Por otra parte, cada planeta está compuesto por varios elementos que definen las características y la naturaleza del mismo:

- **Recursos naturales:** el conjunto de los recursos naturales y las características que tiene el planeta; es decir, agua, terreno transitable, minerales, inestabilidad climática y cantidad de criaturas salvajes.
- **Unidades:** se refiere a los elementos que se crean o colocan en un mundo por la acción del jugador. Se catalogan en unidades industriales (armamento y fábricas) y en unidades vivas (solados y trabajadores).

Durante la partida se pueden desatar procesos "evolutivos" que modifiquen la composición de los recursos y las unidades en cada planeta, esto se explica detalladamente en el próximo capítulo, pues es parte de las reglas lógicas que se pueden configurar en el editor.

Al inicio de la partida, cada jugador tendrá un conjunto de unidades bajo su mando, y podrá manejarlo en su estrategia de juego mediante la acción de enviar misiones a diferentes planetas. Para cada misión se define el mundo de origen, el mundo destino, y número de unidades que se enviarán de cada tipo. Los tres tipos de misiones que se pueden enviar son:

- **Conquista:** cuando se envían tropas a un mundo que no tiene dueño.
- **Invasión:** se envía una misión a un planeta que es propiedad de otro de los jugadores.
- **Expansión:** se envían tropas a un mundo que es propiedad del jugador que envía la misión, es decir, el planeta origen y el destino pertenecen al mismo jugador.

En cada momento, el *bot* debe elegir el movimiento que realizará a partir del estado del mapa que le va suministrando el *Game_Engine*. El tiempo que tarda una misión en llegar a su planeta destino es directamente proporcional a la distancia entre el planeta de origen y el destino. Cuando la misión llega a un planeta enemigo, tiene lugar un enfrentamiento y gana el que más tropas tenga (es decir, el invasor o el planeta destino), si gana el invasor ese planeta pasa a ser propiedad del jugador emisor de la misión. En caso de que el planeta destino sea del propio jugador, ambas flotas se unen, sumando sus naves. El juego finaliza si un solo jugador es el dueño de todas las conquistas del mapa o si se sobrepasa un tiempo límite de partida y en ese caso, gana el que más conquistas tenga o termina en empate si los jugadores tienen el mismo número de planetas conquistados.

Capítulo 3

Editor de reglas

En este capítulo se explicarán los componentes del editor de reglas desarrollado para el presente proyecto; se hará referencia desde el análisis de requisitos hasta la descripción de las clases más relevantes, incluyendo el formato diseñado para representar la lógica del juego y los diagramas más relevantes que muestran su funcionamiento.

3.1 Conceptualización de la solución

Inicialmente el juego RTS de *Eryna* tenía su lógica insertada dentro del *engine*, por lo que se le daba al juego un comportamiento estático, sin variaciones. Esta situación suponía un problema a la hora de implementar el editor de reglas, ya que éstas no admitían edición dinámica. Por ello, en primer lugar se llevó a cabo un rediseño de la arquitectura del juego (separando la lógica del resto del juego) para posibilitar la parametrización de sus reglas pudiendo, así, modificarlas de forma sencilla.

El editor de reglas conforma un sistema independiente del proyecto Eryna pero que se relaciona estrechamente con él, permitiendo generar y editar parte de las reglas y recursos del juego (ver Figura 2.6 del Capítulo 2). De esta forma, el usuario (investigador) que use el editor de reglas generará un fichero con los parámetros que haya introducido en él y dicho fichero será el que utilizará el módulo *Game_Resource* de *Eryna* para modificar sus reglas.

3.2 Análisis de requisitos

En esta sección se especifican los requisitos funcionales y no funcionales que se definieron para guiar el desarrollo de este proyecto y plasmar de forma objetiva el alcance de la solución. Adicionalmente, se muestra un diagrama de casos de uso del programa.

3.2.1 Requisitos funcionales

- **RF1:** El usuario debe ser capaz de guardar un fichero con valores para las reglas del juego.
- **RF2:** El usuario debe poder editar las reglas del juego RTS mediante la aplicación.
- **RF3:** El usuario debe ser capaz de cargar un fichero a la aplicación con valores para las reglas del juego.
- **RF4:** El usuario debe poder crear nuevas misiones a su propio gusto.
- **RF5:** El usuario no debe poder editar los nombres ni las descripciones de las misiones base del juego; es decir, Conquista, Invasión y Expansión.

- **RF6:** El usuario debe poder editar la cantidad máxima de unidades que se pueden enviar en las misiones.
- **RF7:** El usuario debe ser capaz de seleccionar el tipo de batalla que se librará en el juego cuando las misiones de varios jugadores coincidan en un mismo planeta.
- **RF8:** El usuario debe ser capaz de editar los valores de los recursos industriales y bióticos del juego.
- **RF9:** El usuario debe poder editar los valores de los factores evolutivos de los recursos naturales, seres vivos y unidades industriales.
- **RF10:** El usuario debe poder activar y desactivar la evolución de los factores evolutivos que desee.
- **RF11:** El usuario debe ser capaz de habilitar el cansancio de vuelo para las tropas vivas y de editar el valor del factor de cansancio.

3.2.2 Requisitos no funcionales

- **RNF1:** La aplicación debe informar al usuario cada vez que realice alguna acción de importancia con su resultado.
- **RNF2:** La aplicación debe asegurarse de que el usuario ha revisado todas las reglas antes de guardarlas en un fichero.
- **RNF3:** La aplicación debe mostrar detalles de los errores cada vez que el usuario introduzca algún dato no válido.
- **RNF4:** La aplicación debe soportar el inglés como idioma además del español.
- **RNF5:** La aplicación debe disponer de un sistema de ayuda avanzada al usuario sobre las reglas que puede editar.
- **RNF6:** La aplicación debe ser fácil y rápida de manejar.
- **RNF7:** La aplicación debe ocupar poco espacio y ejecutarse rápido.

3.2.3 Diagrama de casos de uso

Un diagrama de casos de uso permite detallar de forma visual los requisitos funcionales del sistema que se desee implementar y disponer de una visión de él como caja negra. Esto es de alta importancia a la hora de desarrollar un nuevo software ya que permite tener una especificación abstracta del mismo, permitiendo hallar errores en el flujo del sistema antes de implementarlo.

En el caso del editor de reglas desarrollado para este proyecto, el diagrama de casos de uso es el que aparece como Figura 3.1. En este diagrama están representados todos los requisitos funcionales descritos en el apartado 3.2.1, aunque algunos de ellos están representados por un mismo caso de uso.

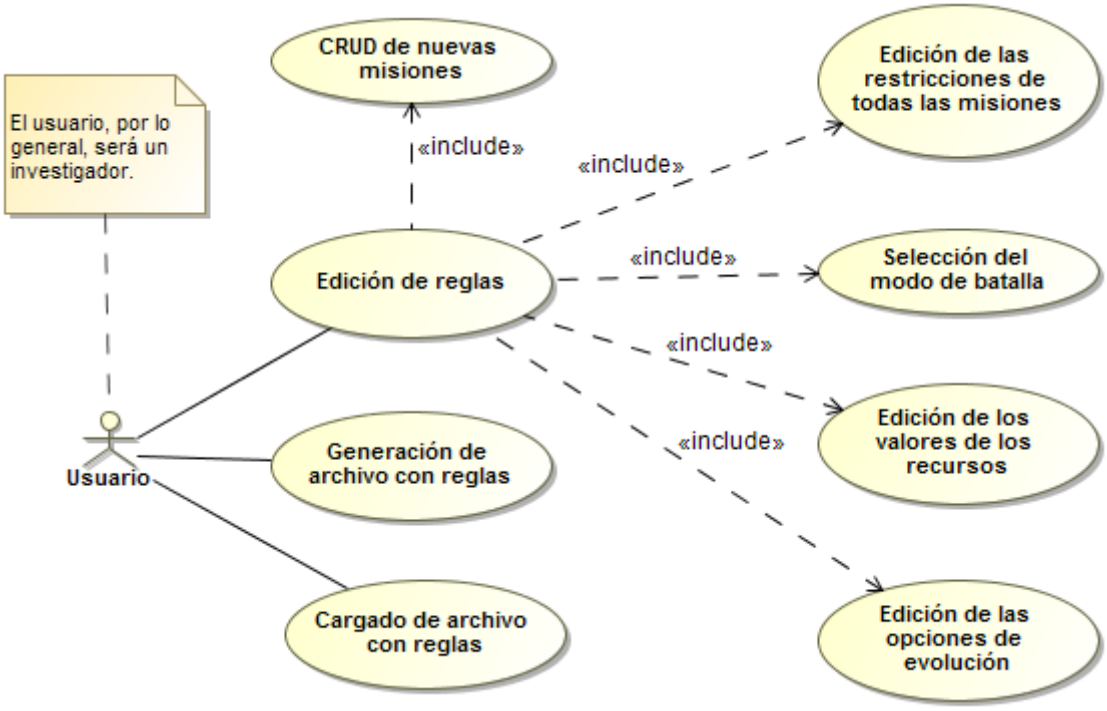

Figura 3.1. Diagrama de casos de uso del editor de reglas

A continuación se pormenorizan las relaciones entre requisitos funcionales y los casos de uso expuestos:

- RF1 → Generación de archivo con reglas
- RF2 → Edición de reglas
- RF3 → Cargado de archivo con reglas
- RF4; RF5 → CRUD de nuevas misiones
- RF6 → Edición de las restricciones de todas las misiones
- RF7 → Selección del modo de batalla
- RF8 → Edición de los valores de los recursos
- RF9; RF10; RF11 → Edición de las opciones de evolución

Debido a que el cansancio de vuelo se trata de una evolución de los valores de las tropas mientras vuelan éste se ha incluido dentro del caso de uso “Edición de las opciones de evolución”. Así mismo, el caso de uso “CRUD de nuevas misiones” implica que no hay un CRUD de las misiones ya existentes y, por tanto, este caso de uso también cubre al requisito RF5 además de al RF4.

De esta forma todos los requisitos funcionales de la aplicación quedan completamente cubiertos y ya se puede proceder a definir los detalles de implementación de la misma.

3.3 Formato para las reglas de juego

El programa requiere de un archivo en el que guardar y desde el que cargar las reglas del juego, esto quiere decir que es necesario encontrar un formato que no sea demasiado complejo para que los investigadores que así lo deseen los puedan modificar directamente sin necesidad de usar el editor.

Los lenguajes de descripción de juegos o GDL (*Game Description Language*, por sus siglas en inglés) son altamente útiles en este aspecto ya que nos permiten detallar las reglas de los juegos de forma unívoca y eficaz [Mahlmann, 2011]. Como ejemplos de estos lenguajes se pueden destacar varios con diferentes enfoques. El *Stanford GDL* es un lenguaje relativamente independiente de los géneros pero está limitado a juegos con espacio de estado discreto y sus descripciones tienen a ser bastante largas debido a que se basa en la lógica de primer orden [Love, 2008]. El *Ludi GDL* propuesto por Cameron Browne, en contraposición al *Stanford GDL*, se limita a juegos de mesa de 2 jugadores con restricciones de elementos y tableros para ser mucho más conciso en las descripciones [Browne, 2008]. Por último cabe mencionar el *Ludocore* de Smith y Mateas, el cual expresa juegos *arcade* en 2D usando programación lógica [Smith, 2010].

El GDL más interesante de todos para este proyecto podría ser el propuesto por Mahlmann et al: SGDL (*Strategy Game Description Language*, por sus siglas en inglés) [Mahlmann, 2011]. Este lenguaje está completamente centrado en los juegos RTS y es relativamente fácil de leer por un humano. La desventaja principal es que para el tipo de reglas que se necesitan especificar en este proyecto es un tanto engorroso. No obstante, Mahlmann et al. especifican para su SGDL tres capas en las que se pueden dividir los juegos de tipo RTS:

- **Capa de mecánicas:** determina las reglas fundamentales del juego, como por ejemplo los acciones de ataque o el tipo de escenario que usa el juego.
- **Capa de ontología:** especifica los elementos clave que pueden existir en el juego y sus propiedades.
- **Capa de instancia:** detalla la organización específica de un mapa o de un nivel del juego.

Siguiendo esta definición de las capas, este proyecto solo se centrará en la capa de mecánicas, ya que el objetivo es modificar las mecánicas de juego mediante el editor y, luego, generar diferentes variaciones de las mecánicas mediante un algoritmo evolutivo.

Debido a que se necesita utilizar un lenguaje de descripción de reglas más simple que SGDL pero igual de eficaz e igual de fácil de leer a simple vista por un humano, se ha optado por seguir los pasos del sistema *ANGELINA* [Cook, 2011] para este proyecto. *ANGELINA* fue desarrollado por Michael Cook y Simon Colton y sirve para crear juegos de género *arcade* desde cero; creando las reglas, los mapas de terreno y la posición de los jugadores. Para este propósito, usaron el formato XML para representar las reglas y las características de los juegos (ver Figura 3.2).

```
<scorelimit>89</scorelimit>
<timelimit>21</timelimit>
<redmovement>RANDLONG</redmovement>
<greenmovement>CLOCKWISE</greenmovement>
<bluemovement>CLOCKWISE</bluemovement>
<rules>
  <rule>PLAYER,BLUE,NOTHING,NOTHING,1</rule>
  <rule>PLAYER,RED,DEATH,DEATH,0</rule>
  <rule>PLAYER,NONE,DEATH,TELEPORT,-1</rule>
  <rule>OBSTACLE,RED,NOTHING,TELEPORT,-1</rule>
</rules>
```

Figura 3.2. XML de especificación de reglas del sistema *ANGELINA*

De esta forma, los archivos que produce (y que puede cargar) el editor de reglas para *Eryna* serán del formato XML y del formato JSON, ya que ambos son muy estructurados, extensibles y fáciles de leer por un ser humano a simple vista. Los ficheros XML y JSON correspondientes a los valores predeterminados de las reglas del juego se encuentran en los Anexos 1 y 2, respectivamente, de este mismo documento.

3.4 Reglas de juego

Ahora que se ha determinado el formato para las reglas del juego, resulta imprescindible definir cuáles de ellas se usarán para modificarlas y cuáles deben ser sus rangos de variabilidad para el editor de reglas (recordemos que el usuario que así lo desee siempre tiene la opción de editar manualmente los ficheros XML y JSON bajo su propia cuenta y riesgo). Con la inclusión de rangos se pretende acotar el alcance de los cambios en las reglas para que éstos no conduzcan a situaciones indeseadas y, además, para facilitar al usuario el uso de la herramienta de edición como forma de abstracción de las reglas en sí mismas.

Las reglas escogidas se dividen en varias categorías atendiendo a la función que cumplen dentro del juego. Cada una de estas categorías se asocia a un *panel* diferente de la aplicación, de tal modo que esta separación por funciones también es visible para el usuario. Las categorías disponibles y sus correspondientes reglas son las que se especifican a lo largo de las siguientes secciones.

3.4.1 Misiones

La primera de las categorías de reglas del juego son las llamadas misiones: acciones de envío de tropas de un planeta a otro con diferentes funciones y restricciones. El juego dispone de 3 tipos de misiones base:

- **Conquista:** acción de enviar tropas a un planeta que carece de dueño para adueñarse de él.
- **Invasión:** acción de enviar tropas a un planeta cuyo dueño es otro jugadores con la finalidad de arrebatárselo.
- **Expansión:** acción de enviar tropas a un planeta aliado con el objetivo de potenciarlo.

Además de estas misiones base, el usuario puede crear y definir todas las misiones adicionales que desee.

Para cada misión, ya sea una base o una creada por el propio usuario, existe la siguiente serie de reglas modificables:

- **Constructores:** cantidad máxima de la unidad tipo constructor que se puede enviar en la misión.
- **Tanques:** cantidad máxima de la unidad tipo tanque que se puede enviar en la misión.
- **Barcos:** cantidad máxima de la unidad tipo barco que se puede enviar en la misión.
- **Aviones:** cantidad máxima de la unidad tipo avión que se puede enviar en la misión.
- **Soldados:** cantidad máxima de la unidad tipo soldado que se puede enviar en la misión.
- **Trabajadores:** cantidad máxima de la unidad tipo trabajador que se puede enviar en la misión.
- **Valores ilimitados:** si se activa esta regla, la cantidad máxima de todas las unidades anteriores que se puede enviar en la misión es ilimitada.

3.4.2 Batallas

Las batallas en el juego se producen cuando se realiza el envío de tropas a un planeta no aliado y la llegada de las mismas coincide con la llegada de tropas de otro jugador (o jugadores) al mismo planeta.

De este modo, una de las reglas modificables del juego es el modo de batalla que éste ejecutará cuando se produzca el caso anterior. El usuario puede escoger entre 4 opciones predeterminadas posibles y no puede crear una por su propia cuenta. Estas opciones predeterminadas son las que se especifican a continuación:

- **El más fuerte:** sólo los dos jugadores más fuertes luchan entre sí y el resto son eliminados del combate.
- **Torneo de eliminación directa:** cada jugador lucha contra otro jugador escogido aleatoriamente. El perdedor de cada encuentro es eliminado del combate y el vencedor lucha contra otro vencedor. Este ciclo se repite hasta que solamente quede un vencedor.
- **Torneo por grupos:** los jugadores son organizados por grupos aleatoriamente. Cada jugador lucha contra los demás jugadores del mismo grupo, ganando puntos en función de los resultados de cada combate. El jugador con más puntos es el vencedor del grupo y, luego, los vencedores se enfrentan entre ellos mediante torneos con el sistema de eliminación directa.
- **Aleatorio:** se seleccionan aleatoriamente dos jugadores para luchar entre sí y el resto son eliminados del combate.

3.4.3 Recursos

A esta categoría pertenecen las reglas relacionadas con los recursos disponibles del juego, los cuales pueden ser bióticos o industriales.

Con respecto a los recursos bióticos, las reglas disponibles son:

- **Fuerza de trabajador:** determina el nivel de fuerza de combate por defecto que tienen las unidades de tipo trabajador.
- **Fuerza de soldado:** determina el nivel de fuerza de combate por defecto que tienen las unidades de tipo soldado.
- **Hambre:** determina el nivel de hambre por defecto de todas las unidades bióticas (trabajador y soldado).
- **Libido:** determina el nivel de libido por defecto de todas las unidades bióticas (trabajador y soldado).

Los valores de hambre y de libido afectan a la eficacia de las tropas en su labor, y los valores de fuerza se usan para los combates (en el caso de los recursos industriales el uso es el mismo).

Los recursos industriales, por su parte, tienen las siguientes reglas modificables a su disposición:

- **Fuerza de máquina constructora:** determina el nivel de fuerza de combate por defecto de las unidades de tipo máquina constructora.
- **Fuerza de nave aérea:** determina el nivel de fuerza de combate por defecto de las unidades de tipo nave aérea.

- **Fuerza de nave acuática:** determina el nivel de fuerza de combate por defecto de las unidades de tipo nave acuática.
- **Fuerza de nave terrestre:** determina el nivel de fuerza de combate por defecto de las unidades de tipo nave terrestre.

3.4.4 Evolución

Por último, en esta categoría se hallan las reglas relacionadas con la evolución de las diferentes unidades del juego (en el caso de las unidades industriales hablamos de desgaste en lugar de evolución). Además, también se pueden modificar las reglas de evolución de los recursos naturales que se encuentran en los planetas.

Las diferentes reglas modificables, tanto para seres vivos como para recursos industriales como para recursos naturales, son las que aparecen especificadas a continuación:

- **Evolución de seres vivos:** permite activar y desactivar la evolución de los diferentes seres vivos en el juego.
- **Factor evolutivo de seres vivos:** si la regla de evolución de los seres vivos está activa, indica el factor de evolución de los mismos.
- **Factor evolutivo de bestias:** si la regla de evolución de los seres vivos está activa, indica el factor de evolución de las bestias neutrales de los planetas.
- **Factor de recuperación:** si la regla de evolución de los seres vivos está activa, indica el factor con el que las unidades vivas reducirán sus niveles de hambre y libido y aumentarán su nivel de fuerza con el paso del tiempo.
- **Cansancio de vuelo:** permite activar y desactiva el cansancio de vuelo para las unidades vivas, lo que afectará a su rendimiento en combate.
- **Factor de reducción de fuerza:** si la regla de cansancio de vuelo está activa, indica el factor de reducción que se le aplicará al valor de fuerza de las unidades vivas durante los vuelos.
- **Desgaste de recursos industriales:** permite activar y desactivar el desgaste de los recursos industriales.
- **Factor de desgaste de recursos industriales:** si la regla de desgaste de recursos industriales está activa, indica el factor de desgaste que sufrirán dichos recursos con el paso del tiempo.
- **Evolución de recursos naturales:** permite activar y desactivar la evolución de los recursos naturales pertenecientes a los planetas.
- **Factor evolutivo de recursos naturales:** si la regla de evolución de recursos naturales está activa, indica el factor con el que evolucionarán los recursos naturales.

3.5 Clases

El editor está programado siguiendo el patrón de arquitectura software de modelo-vista-controlador o MVC. Este patrón resulta muy adecuado para este proyecto porque permite separar los datos (modelo) de la interfaz de usuario (vista) mediante el uso de una clase encargada de comunicar entre sí ambas partes (controlador). De este modo, si en algún momento del futuro es necesario añadir nuevas reglas bastaría con agregar nuevas clases para la vista y el modelo y añadir nuevos métodos al controlador para su intercomunicación, sin afectar por ello al resto de los componentes del editor.

En los siguientes apartados se mostrarán los diagramas de clases correspondientes y se detallará el funcionamiento de cada clase programada.

3.5.1 Diagramas de clases

Los diagramas de clases son muy útiles para mostrar toda la arquitectura de un software, especificando sus clases y las relaciones existentes entre ellas. El diagrama de clases correspondiente al editor de reglas desarrollado se puede observar en la Figura 3.3. El diagrama de clases de la Figura 3.4 es uno más detallado del paquete *views* que aparece en la Figura 3.3 para poder observar las relaciones que hay entre las clases de la interfaz.

3.5.2 Paquete *models*

En este paquete se hallan las clases correspondientes al modelo del software. Todas ellas son del tipo POJO (*Plain Old Java Object*, por sus siglas en inglés), ya que solo implementan una serie de atributos y sus correspondientes métodos *getter* y *setter* (métodos que obtienen y establecen, respectivamente, los valores de los atributos definidos). Estas clases de modelo son las 4 que aparecen a continuación:

- **BattleModeData:** se usa para guardar el código del modo de batalla seleccionado por el usuario.
- **EvolutionsOptionsData:** sirve para guardar los factores evolutivos de los recursos naturales, de los seres vivos (incluyendo el cansancio de vuelo) y de los recursos industriales. Así mismo, también se usa para la activación o desactivación de dichos factores.
- **MissionData:** se usa para guardar las restricciones de una determinada misión, de la cual también guarda su nombre y su información. También sirve para el guardado de la activación o desactivación de restricciones sin límite.
- **ResourcesData:** se utiliza para guardar todos los valores de fuerza de combate de las unidades bióticas e industriales. En el caso de las unidades bióticas, esta clase también sirve para guardar sus valores de libido y hambre.

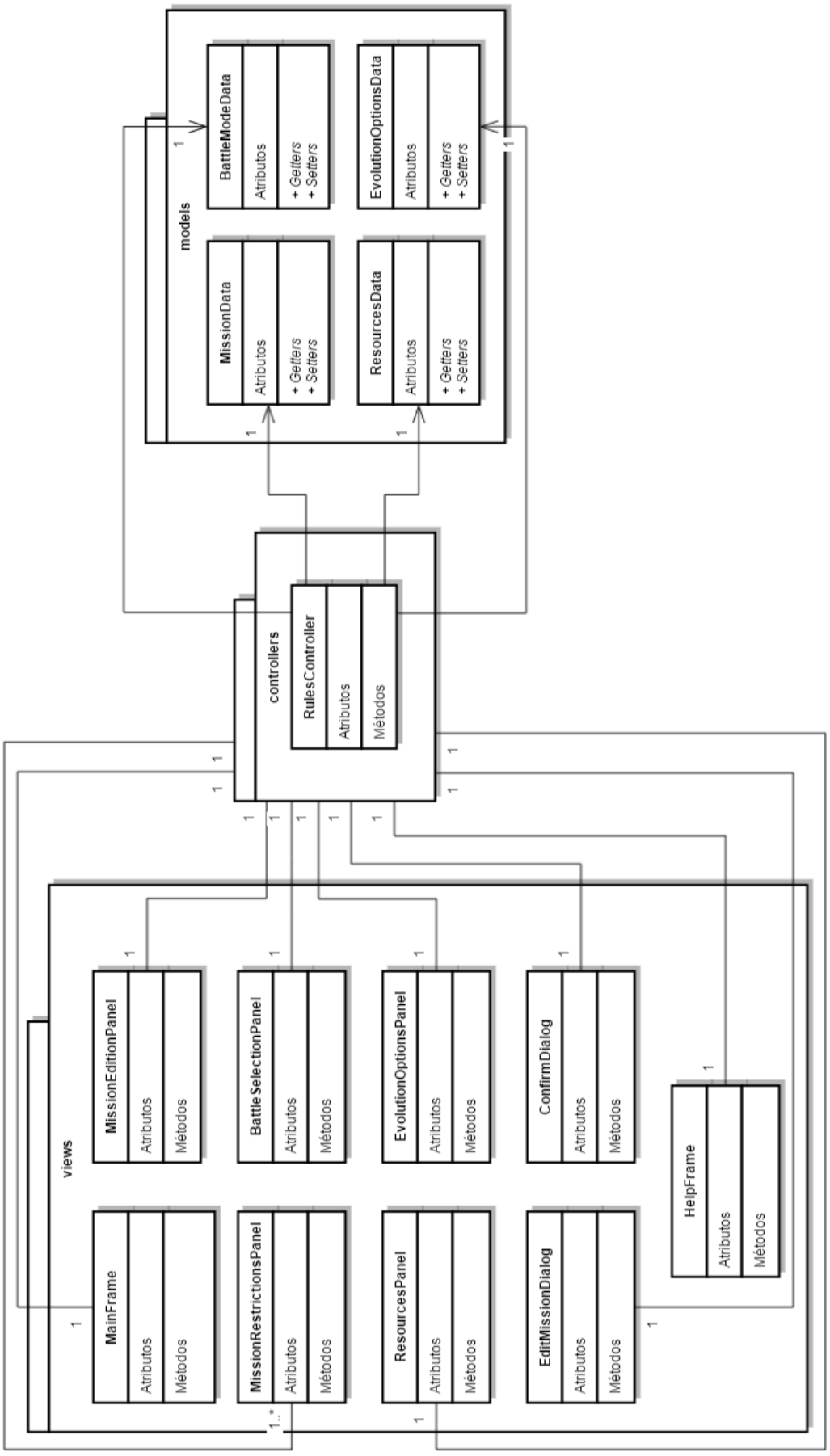

Figura 3.3. Diagrama de clases del editor de reglas

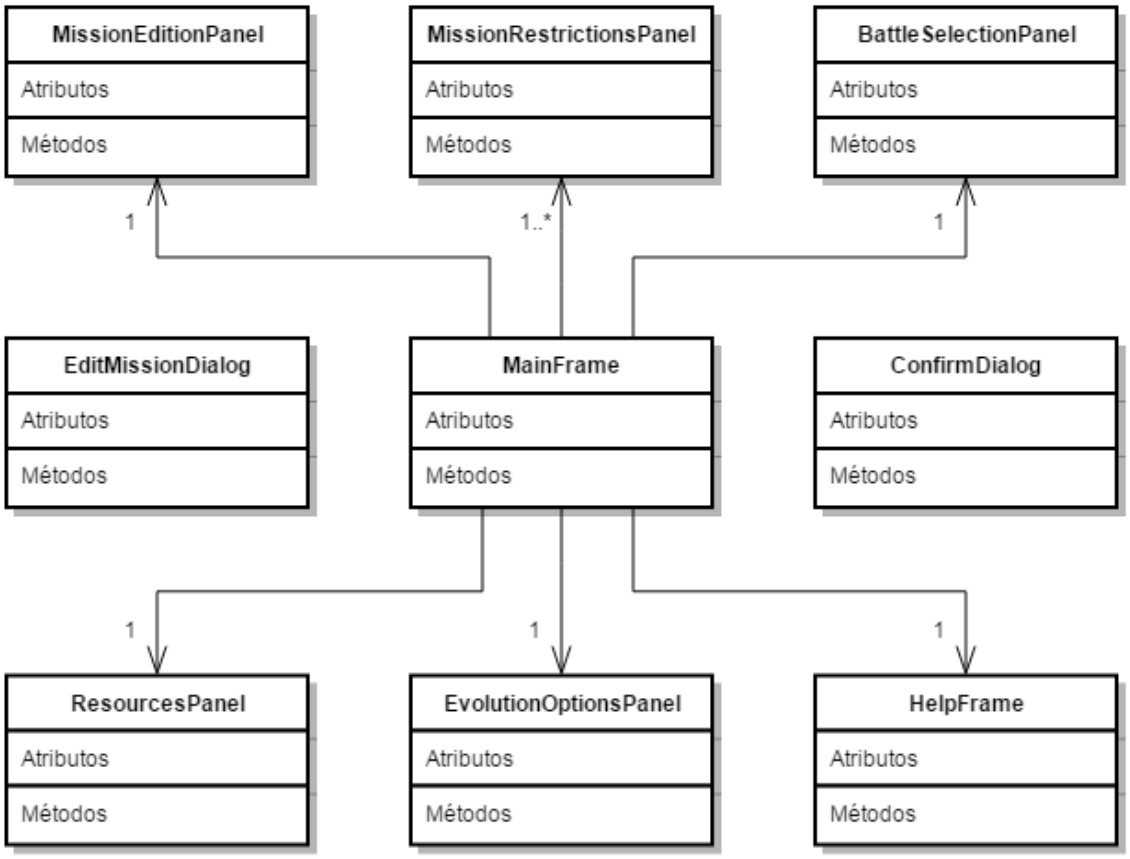

Figura 3.4. Diagrama de clases del paquete *views* del editor de reglas

3.5.3 Paquete *views*

Este es el paquete destinado a las clases de la interfaz o vistas. Estas clases sirven como forma de comunicación con el software y de abstracción del usuario con respecto a los modelos usados por el programa. De esta forma, en algunos paneles se usan escalas numéricas distintas a las que se usan en el modelo para que el usuario las entienda más fácilmente o, incluso en el panel *EvolutionOptionsPanel*, se usan menús desplegables con valores predeterminados entre los que elegir. Los usuarios más avanzados que quieran definir los parámetros de las reglas con mayor precisión siempre tienen la opción de modificar directamente los ficheros XML o JSON.

Hay varios puntos comunes entre las clases de este paquete: todas las clases tienen asignado un controlador (instancia de la clase *RulesController*) y todas tienen un método llamado *setAllLabels()* que permite escribir todos los textos de la interfaz. Además, todos los paneles excepto *MissionEditionPanel* comparten la funcionalidad de bloqueo; es decir, disponen de un botón para bloquear (si no hay errores en los valores introducidos) y desbloquear el panel que permite cumplir con el requisito no funcional RNF2.

Las clases que se encuentran en este paquete son múltiples y de varios tipos diferentes como *panels*, *frames* o *dialogs*:

- **MainFrame:** este *frame* es el principal de la aplicación, en él se incluyen todos los paneles de edición de reglas. Dispone de 4 secciones:
    - Menú: ubicado en la parte superior del *frame*, dispone de las opciones de guardado y cargado de ficheros, cambio de idioma (en cumplimiento del requisito no funcional RNF4) y ayuda al usuario (cumple el requisito no funcional RNF5).
    - Árbol de selección: ubicado en la parte izquierda del *frame*, permite seleccionar los distintos *panels* de edición de reglas que aparecerán en la sección de edición de reglas.
    - Edición de reglas: ubicado en la parte derecha del *frame*, posibilita al usuario editar las reglas basándose en la opción que éste haya escogido en el árbol de selección (se mostrará un *panel* u otro en función de la selección).
    - Mensajes de sistema: ubicado en la parte inferior, muestra todos los mensajes del sistema al usuario tales como acciones realizadas con éxito o mensajes de error (cumpliendo, así, los requisitos no funcionales RNF1 y RNF3).
- **HelpFrame:** este *frame* se lanza cuando el usuario escoge la opción de ayuda en el menú de *MainFrame* y en él se muestra el texto de ayuda en el idioma que esté seleccionado en el *MainFrame*. Este *frame* se lanza en paralelo a *MainFrame* pudiendo el usuario manejar ambos a la vez.
- **BattleSelectionPanel:** este panel permite al usuario seleccionar de entre 4 opciones el modo de batalla que ejecutará el juego en caso de que las misiones de varios jugadores lleguen a la vez a un planeta. Estos modos de batalla disponibles son “el más fuerte”, “torneo de eliminación directa”, “torneo por grupos” y “aleatorio”.
- **EvolutionOptionsPanel:** posibilita al usuario activar y desactivar la evolución de los recursos naturales, de los seres vivos y de los recursos industriales (aunque para éstos se usa el término desgaste en lugar de evolución) y seleccionar valores para sus factores de evolución mediante menús desplegables con valores predeterminados (muy bajo, bajo, medio, alto y muy alto). También permite activar y desactivar el cansancio de las tropas vivas cuando vuelan y modificar su factor de cansancio.
- **MissionEditionPanel:** este panel es el único sin función de bloqueo y dispone de una lista seleccionable con las misiones del sistema en cada momento (ubicada en la parte superior) y un campo donde aparece la descripción de la misión que esté seleccionada en ese momento (ubicado en la parte inferior). Además, posee 3 opciones de acción para el usuario: crear, editar y eliminar. Si crea o edita una nueva misión, se lanzará un *EditMissionDialog* donde podrá introducir los valores pertinentes. Cuando una misión nueva es creada, aparecerá en el árbol de selección de *MainFrame* (para editar sus restricciones) y se actualizará la lista de misiones de este propio panel. Si el usuario selecciona la opción de eliminar una misión se lanzará un *ConfirmDialog* pidiendo la

confirmación al usuario para realizar la acción. En caso de que el usuario confirme la eliminación se actualizará la lista de misiones del panel y el árbol de selección de *MainFrame*. Las opciones de edición y eliminación no se pueden ejecutar sobre las misiones base del sistema (conquista, invasión y expansión).

- **MissionRestrictionsPanel:** permite al usuario definir la cantidad máxima de todos los tipos de unidad (constructores, tanques, barcos, aviones, soldados y trabajadores) que se pueden enviar en la misión que se haya seleccionado en el árbol de selección de *MainFrame*. También da la opción al usuario de no fijar límites a la cantidad de las unidades activando la opción de "Valores sin límite". Si no se selecciona esta opción, el usuario podrá especificar valores entre 0 y 100000, ambos inclusive.
- **ResourcesPanel:** este *panel* posibilita al usuario editar los valores de fuerza de todas las unidades del juego y del hambre y la libido por defecto de las unidades bióticas. Todos los valores son expresados en porcentajes para su fácil entendimiento por parte del usuario.
- **ConfirmDialog:** este *dialog* aparece cuando el usuario trata de eliminar una misión que ha creado anteriormente. Dispone de los botones para confirmar la eliminación de la misión del sistema y para cancelar la acción.
- **EditMissionDialog:** este *dialog* aparece cuando el usuario quiere crear una nueva misión o editar una misión ya existente (que no sea una de las misiones base del juego). Dispone de 2 campos: uno para especificar el nombre de la misión y otro para especificar su descripción. Si el usuario está creando una misión nueva estos campos aparecerán vacíos y si está editando una ya existente aparecerán con los valores actuales de dicha misión. En el caso de una nueva misión, no se puede especificar un nombre de misión que ya exista en el sistema.

3.5.4 Paquete *controllers*

Este paquete solo dispone de una clase, *RulesController*, la cual se encarga principalmente de administrar las comunicaciones entre las clases de la vista y las del modelo. Entre sus demás cometidos se encuentran algunos como manejar los cambios de idioma o la carga y guardado de ficheros.

En líneas generales, el controlador dispone de métodos que realizan las siguientes acciones:

- Asignación con vistas y modelos.
- Cargado y guardado de ficheros (XML y JSON).
- Comunicaciones entre vistas y modelos.
- Envío de mensajes de sistema a *MainFrame*.
- Creación de *dialogs* en los momentos adecuados.
- CRUD de misiones.
- Cambio del *locale* de la aplicación (cambio de idioma).

3.5.5 Otros paquetes

En la aplicación existen otros 2 paquetes adicionales a los anteriormente expuestos: *utils* y *resources*.

El paquete *utils* contiene clases que proporcionan diversas utilidades como limitar la cantidad de caracteres que se pueden introducir en los campos de texto, comprobar si una cadena de texto es un cierto tipo de número o la administración de los diferentes valores para los factores de evolución.

Por su parte, el paquete *resources* contiene todos los recursos necesarios para la ejecución de la aplicación tales como ficheros de idioma, iconos y ficheros de ayuda.

3.6 Análisis del espacio de estados

Ahora que han sido definidas todas las reglas del juego que van a poder ser modificadas y sus diferentes acotaciones en el editor de reglas, se puede realizar un análisis del espacio de estados de la aplicación.

A fin de simplificar este análisis no se tendrán en cuenta las posibles nuevas misiones que el usuario pueda crear (no existe un límite en la cantidad de las mismas). Así mismo, tampoco se tendrán en cuenta todos los diferentes valores que el usuario pueda introducir manualmente en los ficheros de reglas: simplemente el análisis se centrará en todas posibilidades del editor.

En primer lugar, se dispone de 3 misiones: Conquista, Invasión y Expansión. Para cada una de ellas existen 6 parámetros para las diferentes unidades cuyos rangos de valores [0, 100000]. Adicionalmente a estos parámetros, también está el parámetro de valores ilimitados con 2 posibles estados: activo o inactivo. Por tanto, hay una cantidad $100001^6 + 1$ de posibles estados para cada misión y una cantidad $(100001^6 + 1)^3$ de estados posibles entre todas las misiones.

Para los modos de batalla no es necesario cálculo alguno de estados: sólo hay 4 posibles estados, ya que se trata de una selección simple entre 4 opciones.

En el caso de las reglas sobre recursos, el editor permite configurar 8 parámetros de tipo porcentual, por lo que la cantidad de estados posibles aquí asciende a un total de 101^8.

Por último, las reglas relacionadas con la evolución de los elementos del juego disponen de 4 parámetros principales con 2 estados (activo e inactivo) y, en función del estado en que se encuentre cada una, se habilitan nuevas opciones (cada una de ellas con 5 estados posibles a su vez). Para calcular correctamente la cantidad de estados disponibles es necesario hacer un análisis caso a caso:

- Evolución de recursos naturales: $5 + 1$ estados
- Evolución de seres vivos: $5^3 + 1$ estados
- Desgaste de recursos industriales: $5 + 1$ estados
- Cansancio de vuelo: $5 + 1$ estados

Estos cálculos permiten concluir que la cantidad total de estados posibles para la configuración de reglas de evolución es de $(5^3 + 1) \cdot 6^3 = 27216$.

Finalmente, para hallar todos los estados posibles del editor, se multiplican todos los estados posibles de cada una de las categorías del editor y el resultado aproximado es $1.18 \cdot 10^{111}$. Es, por tanto, extremadamente complicado evaluar cada uno de los estados posibles del editor (incluso sin tener en cuenta la posibilidad de crear nuevas misiones).

Capítulo 4

Algoritmos genéticos

Los algoritmos genéticos han cobrado una gran importancia en la actualidad debido a su aplicación como técnicas para la resolución de problemas complejos. Es por ello que han sido usados en una enorme cantidad de ocasiones en el mundo de la ingeniería: diseño de circuitos, reconocimiento de patrones, secuenciación de operaciones, programación de rutas, ingeniería aeroespacial, interpolación de superficies, transporte de materiales y un largo etcétera.

A lo largo de este capítulo se detallarán los algoritmos genéticos, especificando su funcionamiento general y sus principales características. Adicionalmente, se definirán los parámetros del algoritmo genético que se usará para realizar la experimentación del presente proyecto: generar configuraciones de reglas de juego aleatorias.

4.1 Funcionamiento de los algoritmos genéticos

Los algoritmos genéticos están basados en el proceso genético de los organismos vivos: a lo largo de las generaciones, las poblaciones de individuos evolucionan en la naturaleza conforme a los principios de la selección natural y la supervivencia del más fuerte, postuladas por Charles Darwin en 1859 en su libro *El origen de las especies*. Desde el punto de vista algorítmico, esta evolución consiste en un proceso iterativo que se repite constantemente hasta que se cumple una determinada condición de parada, la cual puede ser que se haya llegado a un número concreto de iteraciones o que se haya encontrado un individuo ideal (ver Figura 4.1).

Durante la primera generación de inicializa la población de individuos, la cual está formada por un conjunto de cromosomas. La función de éstos es representar las posibles soluciones del problema y se inicializan mediante procesos aleatorios que aportan una diversidad suficiente a la población (por ejemplo, con una distribución de probabilidad uniforme), de forma que la ejecución del algoritmo sea exitosa. En algunos problemas es posible realizar una criba de la población inicial para eliminar los individuos no factibles.

Seguidamente es necesario especificar una función de evaluación de individuos, la cual asignará a cada individuo de la población una medida de su eficacia (*fitness*) para solucionar el problema. Esta medida de eficacia se debe codificar de tal forma que se pueda comparar con la de otro individuo cualquiera simplificando, así, la toma de decisiones del algoritmo. Para obtener esta medida es suficiente con que la función de evaluación sea capaz de decodificar los cromosomas de cada individuo, sustituyendo por ellos las variables del problema a resolver.

A lo largo de la ejecución de un algoritmo genético se debe comprobar la condición de parada tras cada nueva generación de individuos. Por regla general, se suele tomar como resultado del algoritmo al mejor de los individuos obtenido en la generación actual, y se comprueba si éste es suficientemente adecuado comparándolo con la condición de parada. En caso de ser suficientemente adecuado, este individuo será la solución del algoritmo genético. En caso negativo, el algoritmo continuará su ejecución iterativa y se crearán nuevos individuos.

Si es necesario generar una nueva población (porque no se haya cumplido la condición de parada), el algoritmo hará una selección de los padres; es decir, de los individuos que se reproducirán. Esta selección se determina de forma probabilística y en función de su *fitness*. La reproducción es el método mediante el cual se generan nuevos individuos a partir de los padres, pero diferentes de éstos. Existen dos métodos principales de reproducción:

- **Cruce:** consiste en realizar una combinación de parte del código genético de los padres.
- **Mutación:** modifica aleatoriamente uno o varios genes del nuevo individuo, introduciendo nuevo material genético en la población.

Estos nuevos individuos reemplazarán a algunos de los individuos de la población original atendiendo a diversos criterios: reemplazo por selección, reemplazo de población completa, etc.

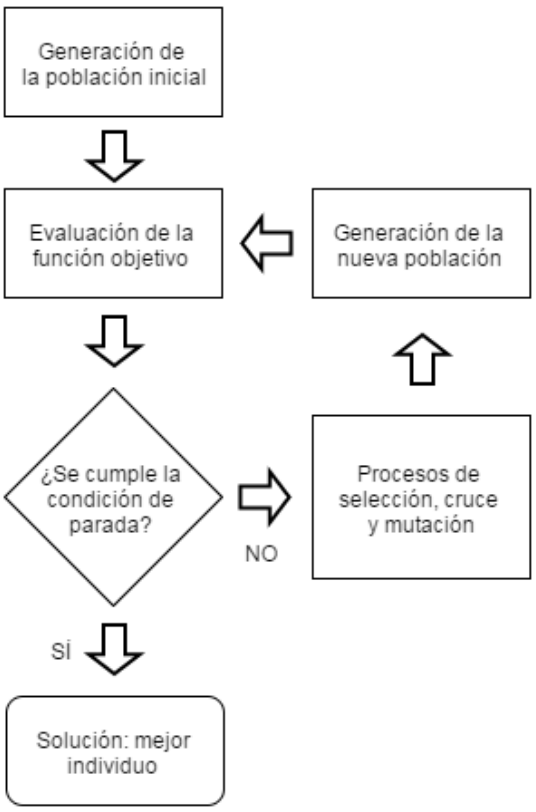

Figura 4.1. Diagrama de flujo de un algoritmo genético

4.2 Tipos de algoritmos genéticos

Los algoritmos genéticos disponen de diferentes implementaciones, cada una basada en una metáfora distinta de la naturaleza:

- **Generacional:** este es el tipo de algoritmo genético considerado como estándar. Su funcionamiento se basa en la forma de reproducción de los insectos, en la que una generación pone huevos, desaparece y es sustituida por una nueva generación. Este tipo de algoritmo utiliza, por tanto, un reemplazo de población completa.
- **Estado estacionario:** este tipo se basa en el esquema generacional de los animales de larga vida (como los mamíferos), en el que los padres coexisten con sus descendientes. De este modo, en este algoritmo es necesario usar un reemplazo por selección o aleatorio, pero nunca de población completa.
- **Paralelo:** se basa en el concepto de que varias poblaciones de individuos aisladas unas de las otras son capaces de evolucionar de forma diferente. Este concepto da lugar a dos modelos diferentes de algoritmo genético paralelo: el de islas y el celular [Alba, 1999].
    - **Modelo de islas:** la población original está dividida en subpoblaciones que evolucionan de forma independiente, mediante el uso de algoritmos genéticos generacionales. De manera ocasional, se producirán migraciones entre las subpoblaciones que propiciarán intercambio genético entre las mismas. Estas migraciones deben estar muy controladas porque si emigran demasiados individuos se pueden acabar eliminando las diferencias genéticas locales.
    - **Modelo celular:** la población se reparte a lo largo de una matriz donde cada individuo solo puede reproducirse con los individuos que tenga alrededor suya (de forma aleatoria o escogiendo los más aptos) y los descendientes ocuparán una posición cercana. De este modo se producen diferencias evolutivas cada cierta distancia entre individuos (si están muy alejados es prácticamente imposible que su material genético se cruce).

4.3 Operadores genéticos

Para propiciar el paso de una generación a la siguiente se aplican diferentes operadores genéticos, entre los que cabe destacar selección, cruce, mutación y reemplazo, por ser los más relevantes y usados.

4.3.1 Selección

El proceso de selección discierne qué individuos poseerán oportunidades para reproducirse y cuáles no. Puesto que los algoritmos genéticos se basan en el proceso de selección natural, los individuos con un mayor valor de *fitness* son los que más posibilidades de reproducirse han de tener. Sin embargo, los individuos menos aptos también deben tener algunas oportunidades ya que, de lo contrario, la población perdería muy rápidamente su diversidad genética.

Hay diferentes tipos de selección disponibles, entre los que destacan:

- **Selección elitista:** siempre se asegura la selección de los individuos más aptos (con un valor de *fitness* mayor) de la población en cada generación.
- **Selección proporcional:** se seleccionan los individuos de forma aleatoria, pero a mayor *fitness* mayor probabilidad de ser escogido.
- **Selección por torneo:** se escoge al azar un número de individuos de la población (tantos como participantes necesite el torneo) y se seleccionan de entre ellos los que dispongan de mayor *fitness* para formar parte de la nueva generación (la cantidad de ganadores se establece previamente).

Dependiendo de qué método se use se obtendrán unos resultados u otros: si se ejerce una alta presión de selección (por ejemplo, selección elitista de pocos individuos o torneos de 2 individuos) la búsqueda de la solución se centrará en un entorno próximo a las mejores soluciones actuales y, si se ejerce una presión baja, se podrán explorar nuevas zonas del espacio de búsqueda.

4.3.2 Cruce

La operación de cruce consiste en la recombinación de los individuos previamente seleccionados para producir la descendencia que formará la nueva generación.

El cruce de dos padres cualesquiera tiene una tasa de probabilidad, la cual indica si se cruzan los cromosomas de los padres o si se engendran descendientes exactamente igual a ellos. Por tanto, a mayor tasa de probabilidad, más cruces se producirán en una población. Esto es de vital importancia para la evolución de las generaciones de forma que, por regla general, se suele trabajar con una tasa de cruce cercana al 90%.

Los métodos de cruce dependen de la representación de los cromosomas escogida: se suelen utilizar números binarios, enteros o reales. Los principales métodos de cruce para representación binaria son los que se detallan a continuación:

- **Cruce básico o unipunto:** consiste en la selección aleatoria de un punto del cromosoma como punto de cruce. Para el primer hijo se copia la parte anterior al punto de cruce del primer padre y la parte posterior del segundo padre; para el otro hijo se sigue el proceso inverso (ver Figura 4.2).
- **Cruce multipunto o n-punto:** sigue el mismo concepto que el cruce básico pero aumentando la cantidad de puntos de cruce. Ofrece como ventaja que el espacio de búsqueda del problema puede ser explorado con mayor facilidad, pero tiene el inconveniente de que es más probable romper buenos esquemas genéticos. De hecho, el profesor de la *George Mason University* Kenneth A. De Jong demostró que usar 2 puntos de cruce supone una mejora con respecto al cruce básico pero, sin embargo, usar más de 2 puntos no beneficia al comportamiento del algoritmo genético [De Jong, 1975].
- **Cruce uniforme:** se genera cada gen del descendiente en función de una máscara de cruce. Para cada gen de los padres, si en su posición en la máscara de cruce hay un 1, el descendiente copiará ese gen de su primer padre, y si hay un 0 lo copiará de su segundo padre (ver Figura 4.3).

Para las representaciones de cromosomas mediante números enteros o reales se usan los siguientes métodos de cruce:

- **Media aritmética:** el gen del descendiente es la suma de los valores de los genes de sus padres dividida entre dos.
- **Media geométrica:** el gen del descendiente es la raíz cuadrada del producto de los valores de los genes de sus padres.
- **Extensión:** se toma la diferencia existente entre los genes situados en las mismas posiciones de los padres y se suma al valor más alto o se resta del valor más bajo.

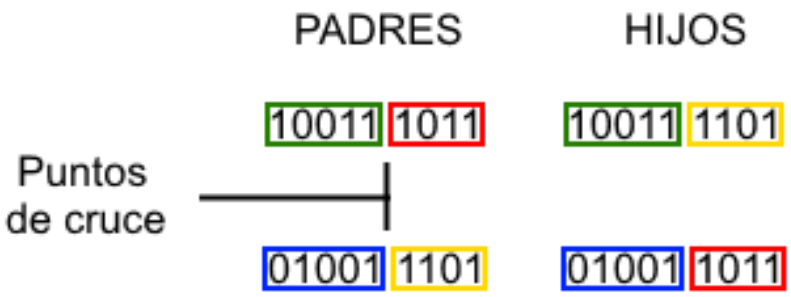

Figura 4.2. Ejemplo de cruce básico (unipunto)

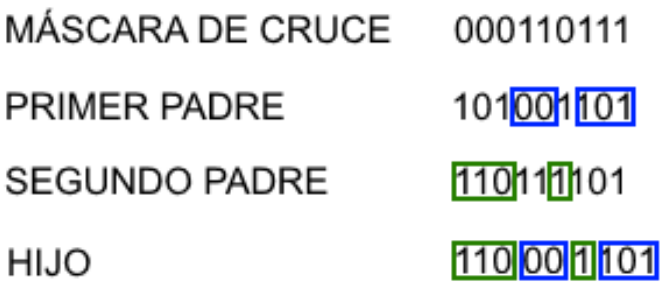

Figura 4.3. Ejemplo de cruce uniforme

4.3.3 Mutación

El proceso de mutación se define como la alteración de los genes de un individuo generado previamente en función de una probabilidad determinada. Su finalidad es añadir variedad genética a los hijos para que éstos no estén limitados solamente a la herencia genética de los padres.

Resulta importante, por lo tanto, escoger bien un valor de probabilidad adecuado a las necesidades del problema. Por lo general, un valor bastante bueno para cromosomas codificados en binario es 1/número de genes (de hecho, este es el valor que se usará más adelante en la experimentación del presente proyecto).

La alteración en los genes producida por una mutación es, en el caso de las codificaciones binarias, la negación del valor original (mutación de 0 a 1 y de 1 a 0). Si la codificación se basa en números enteros o reales, la mutación se puede realizar sumando, restando o multiplicando un valor preestablecido.

4.3.4 Reemplazo

Las técnicas de reemplazo de individuos de la población original por los nuevos generados son las que siguen a continuación:

- **Reemplazo de población completa:** se eliminan todos los padres de la población y ésta es ocupada únicamente por los hijos.
- **Reemplazo por selección:** se reemplazan determinados individuos de la población original por los descendientes atendiendo a dos criterios.
    - **Individuos similares:** se reemplaza un grupo de individuos (normalmente entre 6 y 10) con un *fitness* similar a los de la descendencia.
    - **Individuos menos aptos:** se elimina un porcentaje de los individuos menos aptos de la población original.
- **Reemplazo aleatorio:** el nuevo individuo reemplaza a un individuo de la población original aleatorio.

Capítulo 5

Experimentación

En este último capítulo se expondrán los diferentes experimentos realizados, detallando mediante gráficas los resultados obtenidos. Además, se detallará la codificación del cromosoma para los algoritmos genéticos usados para llevar a cabo estos experimentos.

Para la realización de estos experimentos se ha usado la librería libre *JGAP*, que proporciona un marco de trabajo basado en *Java* sobre algoritmos genéticos y programación genética [JGAP, 2015].

5.1 Algoritmos genéticos para la generación de reglas

Para la elaboración de estos experimentos se han diseñado tres algoritmos genéticos que generan reglas de juego y las evalúan atendiendo a tres criterios diferentes: maximizar la cantidad de turnos en que la partida se considera que está balanceada, maximizar la cantidad de turnos en que la partida se considera que es dinámica y maximizar los dos casos anteriores, dándoles un peso de importancia del 50% a cada uno (las definiciones de partida balanceada y dinámica se detallarán más adelante).

El cromosoma diseñado atiende a las reglas especificadas en el editor de reglas, sin tener en cuenta la creación de nuevas misiones (en el Capítulo 3 quedó demostrada la gran cantidad de estados posibles que existen sólo con las 3 misiones originales). De este modo el cromosoma está compuesto de 40 genes, cada uno representando una regla diferente.

En la Figura 5.1 se pueden observar todos los genes del cromosoma: la primera tabla con los 7 primeros genes corresponde a una de las misiones (los siguientes 14 genes son los mismos, para las otras 2 misiones); luego está el gen correspondiente al modo de batalla, los 8 genes de las reglas de recursos y, por último, los 10 genes finales de las reglas de evolución. En cada uno de ellos se especifica el tipo de datos que generará el algoritmo y el rango de valores posible de esos datos.

Los algoritmos diseñados disponen de los mismos parámetros y operadores genéticos, diferenciándose únicamente en sus funciones de *fitness* que se corresponden con los objetivos mencionados anteriormente. Los parámetros genéticos de ambos son los que se detallan a continuación:

- **Población:** la población de los algoritmos es de 15 individuos creados, inicialmente, de manera aleatoria.
- **Selección:** el proceso de selección se lleva a cabo mediante torneo binario, es decir, un torneo entre 2 individuos.
- **Cruce:** el tipo de operación de cruce es el básico (unipunto), con una tasa de probabilidad del 90%.

- **Mutación:** la mutación se realiza gen a gen con una probabilidad de 1/40, o lo que es lo mismo, estadísticamente 1 de los 40 genes será mutado.
- **Reemplazo:** la población original siempre será reemplazada por la nueva que se haya generado (reemplazo generacional o de población completa).

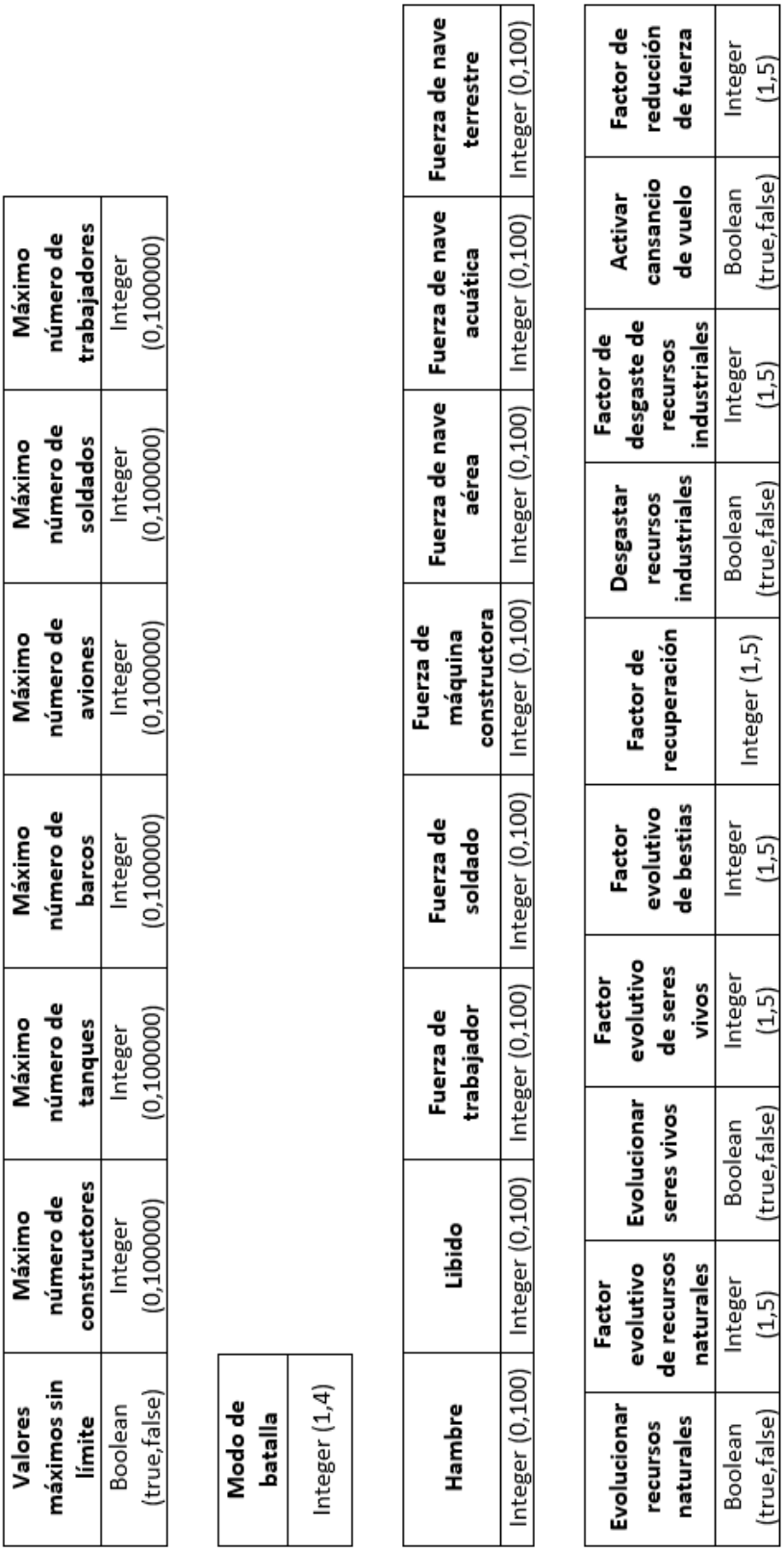

Figura 5.1. Genes del cromosoma.

En la Figura 5.2 se detalla el funcionamiento (en pseudo-código) de los algoritmos implementados para estos experimentos. La función de evaluación cambiará entre balance, dinamismo y balance-dinamismo para cada algoritmo.

```
población ← random (15) ; // La población inicial se genera aleatoriamente
población ← evaluación (población); // Se evalúan los individuos iniciales
i ← 0;
while (i < maxGeneraciones) do
    padres ← selección (población); // Torneo binario
    hijos ← cruce (padres); // Unipunto con 90% de probabilidad
    hijos ← mutación (hijos); // Gen a gen con 2.5% de probabilidad
    población ← reemplazo (hijos); // Reemplazo de población completa
    población ← evaluación (población); // Se evalúa la nueva población
    i ← i + 1;
end while
fittest ← getFittest(población); // Se obtiene el individuo más fuerte de la población
saveFittest (fittest); // Se guardan los datos del individuo más fuerte en un fichero
```

Figura 5.2. Pseudo-código de los algoritmos genéticos.

5.2 Experimentos de reglas generadas genéticamente

Estos experimentos se basan en las tres premisas detalladas anteriormente, las cual persiguen tres objetivos fundamentales:

- Hallar un conjunto de reglas que haga que una partida esté balanceada el máximo tiempo posible.
- Hallar un conjunto de reglas que haga que una partida sea dinámica el máximo tiempo posible.
- Hallar un conjunto de reglas que haga que una partida esté balanceada y sea dinámica al mismo tiempo el máximo tiempo posible.

Que una partida esté *balanceada* quiere decir, en el caso de *Eryna*, que la diferencia entre la cantidad de planetas que poseen los jugadores nunca supere un umbral determinado. En el caso específico de este experimento, el umbral será un 20% de los planetas de los que dispone el mapa. Con respecto al *dinamismo* de la partida, éste se entiende por la cantidad de veces que cambia el jugador que va en cabeza (el que más planetas tiene) a lo largo de la misma. Estos dos objetivos intentan conseguir que las partidas sean divertidas, basándose en una mezcla entre el conocimiento previo del juego y la intuición.

Para obtener unos datos sólidos, se han empleado los mismos *bots* y el mismo mapa en todos los experimentos realizados. Siendo así, los 2 *bots* que se han enfrentado seguían un patrón de ataque completamente aleatorio y el mapa estaba compuesto por 9 planetas (7 neutrales y 1 para cada *bot* al inicio).

5.2.1 Experimento 1: reglas para partidas balanceadas

En las Figuras 5.3 y 5.4 se pueden observar las gráficas obtenidas con este experimento, la primera con la evolución del mayor valor de *fitness* obtenido a lo largo de 30 generaciones de individuos y la segunda con la evolución del *fitness* promedio de cada una de esas generaciones.

En este experimento y los 2 siguientes, los valores de fitness se corresponderán con la cantidad de turnos que cumplan el objetivo propuesto en una partida multiplicados por 100. Esto es para poder tener margen de ajuste de los valores de *fitness* en caso necesario.

En ambas gráficas se puede observar una tendencia al alza de los valores de *fitness*, por lo que se comprueba que el algoritmo genético resulta eficaz a la hora de evolucionar reglas que permitan alargar la cantidad de turnos en los que una partida está balanceada. En el caso de este experimento se alcanzó un máximo de 6 turnos de balance en las partidas, lo cual quiere decir que en 6 turnos de todos los que se disputaron en la partida la cantidad de planetas que poseían los jugadores estaba balanceada.

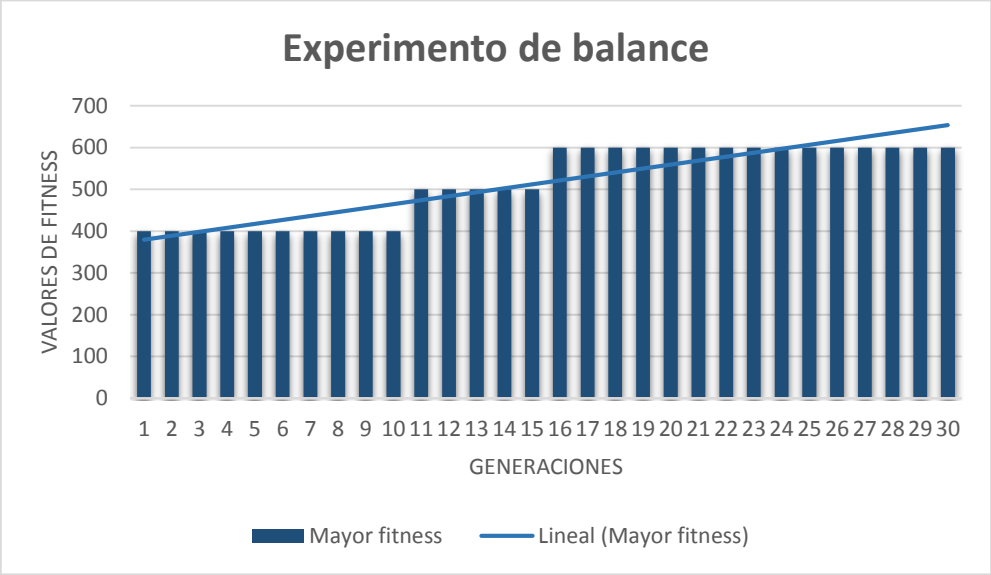

Figura 5.3. Mayor *fitness* de partida balanceada.

Sin embargo, se puede comprobar que en este experimento sólo se han producido 2 mejoras de *fitness* (concretamente, en las generaciones 11 y 16), lo cual es una cifra bastante baja. De todos modos estos datos, aunque no son especialmente buenos, no son preocupantes tampoco, ya que nos encontramos al inicio de una primera fase experimental y es necesario afinar aún los parámetros del algoritmo para obtener unos resultados más satisfactorios.

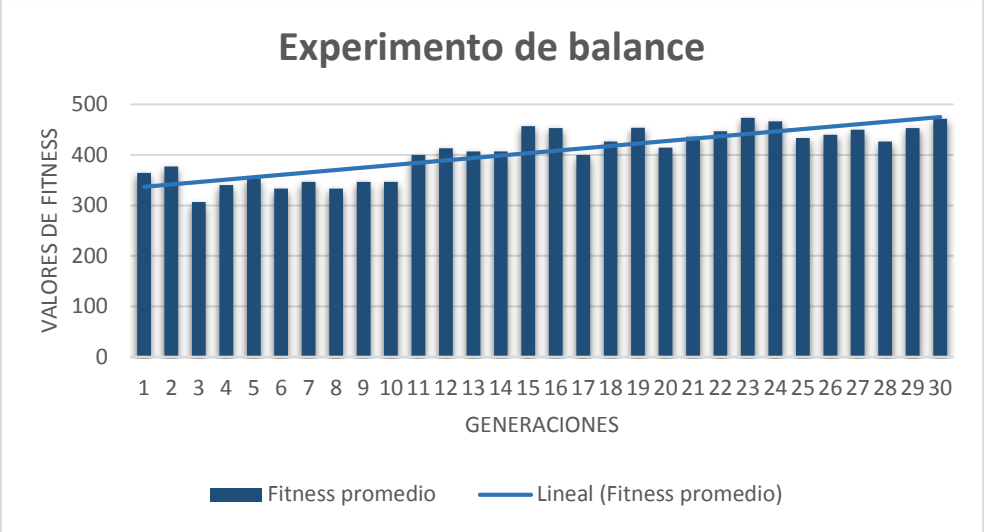

Figura 5.4. *Fitness* promedio de partida balanceada.

5.2.2 Experimento 2: reglas para partidas dinámicas

En las Figuras 5.5 y 5.6 se pueden observar las gráficas obtenidas con este experimento. Análogamente al experimento anterior, la primera gráfica muestra la evolución del mayor valor de *fitness* obtenido a lo largo de 30 generaciones de individuos y la segunda, la evolución del *fitness* promedio de esas generaciones.

Aquí también se observa un claro crecimiento de los valores de *fitness* obtenidos a lo largo de las diferentes generaciones de individuos con los que se ha experimentado. El máximo número de turnos en los que las partidas fueron consideradas dinámicas creció desde los 3 turnos de las primeras generaciones, hasta alcanzar los 9 turnos de las 10 últimas generaciones. Esto quiere decir que se encontró un conjunto de reglas que propiciaron que, a lo largo de una partida, el supuesto ganador de la misma cambiara 9 veces.

Figura 5.5. Mayor *fitness* de partida dinámica.

En este caso, se producen también 2 mejoras del *fitness* únicamente: en las generaciones 3 y 20. Esto, como se comentó en el experimento anterior, es debido a que nos encontramos en una fase muy temprana aún de la experimentación.

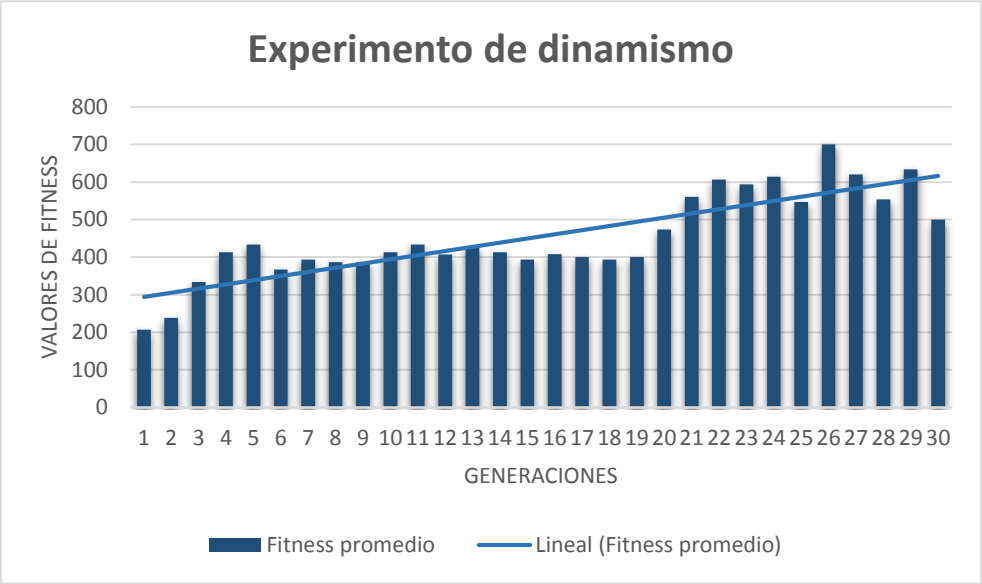

Figura 5.6. *Fitness* promedio de partida dinámica.

Si se examina el *fitness* promedio de cada generación para este experimento se puede comprobar que se produce una fluctuación de su valor mucho mayor que en el experimento de balance de partidas.

5.2.3 Experimento 3: reglas para partidas balanceadas y dinámicas

Por último, en las Figuras 5.7 y 5.8 aparecen las gráficas obtenidas a partir del experimento que trata de maximizar la cantidad de turnos en que una partida está balanceada y, además, es dinámica.

Como en los dos experimentos anteriores, se puede apreciar la evolución ascendente de los valores de *fitness* en cada generación aunque, en este caso, esta evolución es menos pronunciada. El valor máximo de turnos que las partidas están balanceadas y son dinámicas a la vez es 4. A la luz de estos datos, se puede concluir que tratar de encontrar reglas que propicien partidas muy equilibradas y en las que, adicionalmente, se produzcan muchos cambios del supuesto ganador no es tarea fácil y se requeriría una investigación más profunda.

Este se trata de un objetivo bastante más ambicioso que los anteriores y, sin embargo, los resultados obtenidos son muy alentadores y pueden abrir muchas vías de investigación.

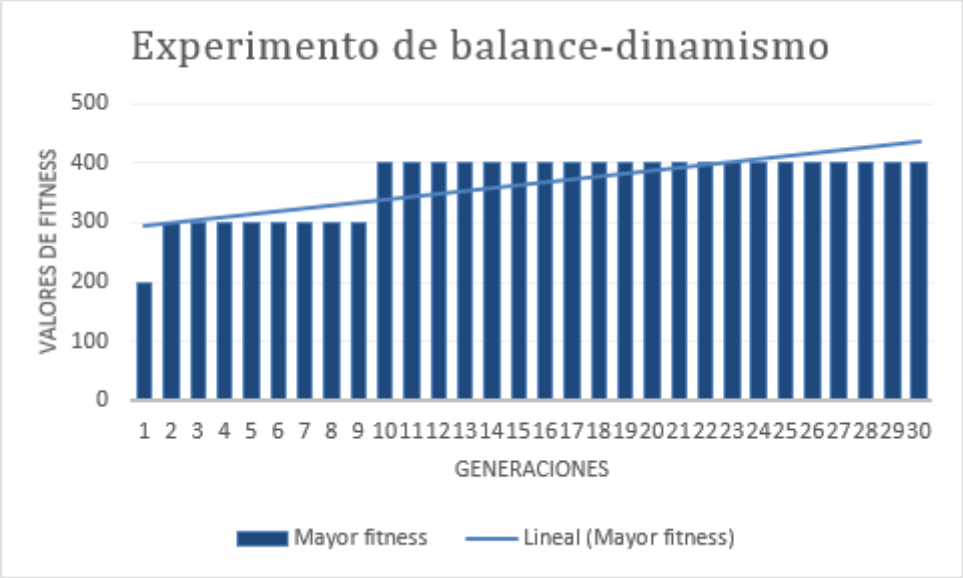

Figura 5.7. Mayor *fitness* de partida balanceada-dinámica.

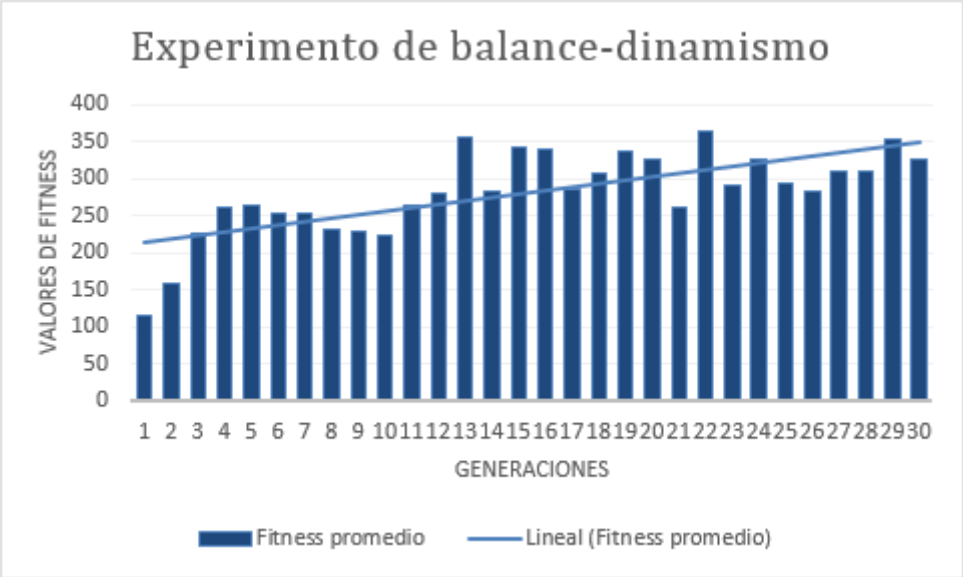

Figura 5.8. *Fitness* promedio de partida balanceada-dinámica.

Conclusiones

Desde el punto de vista del desarrollo del proyecto, se han logrado cumplir los objetivos marcados con cierto grado de éxito. Se propuso realizar una aplicación de escritorio con la que editar las reglas de *Eryna* de una manera fácil e intuitiva y se ha cumplido con creces, permitiendo a los investigadores abstraerse de la implementación del juego para estudiar sus reglas y la influencia que tienen éstas sobre el juego al modificarlas. Adicionalmente, se propuso la implementación de algoritmos genéticos que generasen reglas de juego para poder experimentar con ellas y observar su comportamiento. Este objetivo se ha cumplido con 3 algoritmos que persiguen objetivos diferentes, aunque son necesarios experimentos más exhaustivos en trabajos futuros.

Desde el punto de vista personal y profesional, la realización de este proyecto me ha servido para conocer de primera mano y detalladamente el funcionamiento de los algoritmos genéticos y cómo usarlos. Esto lo considero de una gran importancia, debido a que había oído hablar muchas veces sobre ellos pero nunca los había visto ni sabía cómo funcionan y he podido comprobar lo tremendamente útiles que son para resolver cierto tipo de problemas. A partir de ahora son una herramienta más de mi caja de herramientas de ingeniero de software. También cabe destacar que, gracias al desarrollo de la aplicación de escritorio, he podido usar una gran cantidad de los conocimientos adquiridos a lo largo del grado (*diseño de interfaces de usuario, arquitecturas software, programación concurrente, manejo de servidores, captura de requisitos, diseño de diagramas de clases y casos de uso*, por nombrar algunas), lo cual es de agradecer. Por último, y no por ello menos importante, me he introducido en el mundo de la investigación científica debido a que ha sido necesario que me documentara e investigara muchas publicaciones generación de contenido, reglas y demás elementos relacionados con este proyecto y he aprendido, en cierto grado, a escribir textos científicos.

Desde el punto de vista de la dificultad, la elaboración de este proyecto ha sido toda una experiencia. He tenido que aprender y conocer de primera mano las técnicas PCG y su historia leyendo muchos textos científicos en inglés, he tenido que comprender las reglas internas de un juego de corte RTS que me era totalmente ajeno y he tenido que comprender y programar algoritmos genéticos. Además, es necesaria una mención especial, a las dificultades superadas para lanzar partidas durante los experimentos y cruzar el código de los algoritmos genéticos con el del juego.

Durante todo este proyecto se han estudiado las técnicas de generación automática de contenidos, las reglas lógicas de los videojuegos y cómo se pueden unificar ambos. El objetivo final, pero todavía lejano en el tiempo, sería el lograr la generación automática de juegos completos; es decir, poder crear juegos a golpe de *click* de ratón. Esto supone una enorme dificultad porque, como hemos visto, hay sistemas que son capaces de generar reglas de juego pero éstas son extremadamente limitadas y tienen una acotación muy marcada, pero no hay ninguno que sea capaz de generar un juego desde cero con unas

mecánicas más profundas como son las de los juegos de tipo RTS. En este sentido, existen varias publicaciones sobre el tema pero por el momento, no hay ninguna investigación que trate de abrir el camino a la generación automática de reglas de juego y es ahí donde radica la importancia del presente proyecto. Principalmente, era necesaria una herramienta de edición de reglas para facilitar a los investigadores su labor en un entorno como es el juego *Eryna* para recabar información del comportamiento de ciertas reglas y de su influencia sobre el propio juego. Por otro lado, resulta vital el desarrollo de experimentos de generación de reglas, ya que ello nos permitirá configurar las posibles reglas de un juego determinado para que éste tenga el comportamiento que queramos. Es por eso que los experimentos realizados en este trabajo son un primer paso muy importante de un largo camino que aún queda por recorrer.

Finalmente, para trabajos futuros resultaría de un gran interés realizar muchos más experimentos con diferentes condiciones y estudiar los resultados: con *bots* de diferentes comportamientos, con una gran variedad de mapas, aislando grupos relacionados de reglas para evolucionar únicamente esas y estudiar su comportamiento, etc. Incluso, yendo varios pasos más allá, se podría estudiar el resultado de los juegos con jugadores humanos de diferentes perfiles y sobre otros videojuegos totalmente diferentes. En definitiva todavía es muy pronto para la generación automática de videojuegos pero, citando a Séneca: “no hay camino que no tenga fin”.

Referencias

- [AI, 2015] Google AI Challenge. (2015). Obtenido de la web http://aichallenge.org/

- [Alba, 1999] Alba, Enrique. (1999). *Análisis y Diseño de Algoritmos Genéticos Paralelos Distribuidos*. Tesis doctoral, Universidad de Málaga.

- [Avery et al., 2011] Avery, P., Togelius, Alistar, E. & van Leeuwen, R. P. (2011). Computational intelligence and tower defence games. *IEEE Congress on Evolutionary Computation,* (págs. 1084-1091).

- [Bethesda, 2015a] Bethesda Game Studios. (2015). *Fallout 3.* Obtenido de la web https://fallout4.com/games/fallout-3

- [Bethesda, 2015b] Bethesda Game Studios. (2015). *The Elder Scrolls V: Skyrim.* Obtenido de la web http://www.elderscrolls.com/skyrim

- [Blizzard, 2015] Blizzard. (2015). *Starcraft II*. Obtenido de la web http://eu.battle.net/sc2/es/

- [Browne, 2008] Browne, C. (2008). *Automatic generation and evaluation of recombination games.* Ph.D. thesis, Queensland University of Technology.

- [Cook, 2011] Cook, M. & Colton, S. (2011). Multi-faceted evolution of simple arcade games. *IEEE Conference on Computational Intelligence and Games,* 289-296.

- [De Jong ,1975] De Jong, K. A. (1975). An analysis of the behavior of a class of genetic adaptive systems (Doctoral dissertation, University of Michigan). *Dissertation Abstracts International*, *36*(10), 5140B.

- [DEV, 2015] Asociación Española de Empresas Desarrolladoras de Videojuegos y Software de Entretenimiento (2015). *Libro Blanco del Desarrollo Español de Videojuegos.*

- [EA, 2015] EA Games. (2015). *Spore.* Obtenido de la web http://www.ea.com/es/spore

- [Gutierrez et al., 2014] Gutierrez, A., Lara-Cabrera, R., & Fernández, A. J. (2014). Generación automática de contenido para un nuevo juego basado en el problema de los tres cuerpos. *CoSECivi*, (págs. 199-210).

Referencias

- [Hendrikx et al., 2013] Hendrikx, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013). Procedural content generation for games: A survey. *TOMCCAP, 1*.

- [IEEE, 2015] IEEE. (2015). *Official wiki of the IEEE Task Force on Player Satisfaction Modeling.* Obtenido de la web http://gameai.itu.dk/psm

- [JGAP, 2015] JGAP. (2015). Obtenido de la web http://jgap.sourceforge.net/

- [Lionhead, 2015] Lionhead Studios. (2015). *Fable.* Obtenido de la web http://www.lionhead.com/games/fable/

- [Love, 2008] Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M. (2008). *General Game Playing: Game Description Language Specification*.

- [Lucas, 2009] Lucas, S. M. (2009). Computational Intelligence and AI Games: A New IEEE Transactions. *IEEE Transactions on Computational Intelligence and AI in Games, 1*(1), 1-3.

- [Mahlmann, 2011] Mahlmann, T., Togelius, J. & Yannakakis, G. N. (2011). Towards Procedural Strategy Game Generation: Evolving Complementary Unit Types. *EvoApplications (1),* 93-102.

- [Microsoft, 2015] Microsoft Studios. (2015). *Age of Empires.* Obtenido de la web http://www.ageofempires.com/

- [Mojang, 2015] Mojang. (2015). *Minecraft.* Obtenido de la web http://minecraft.net

- [Nintendo, 2015] Nintendo. (2015). *Mario Kart Wii*. Obtenido de la web https://www.nintendo.es/Juegos/Wii/Mario-Kart-Wii-281848.html

- [Nogueira et al., 2014] Nogueira, M., Cotta, C., & Fernández, A. J. (2014). Eryna: una herramienta de apoyo a la revolución de los videojuegos. *CoSECivi*, (págs. 173-184). Barcelona.

- [ORTS, 2015] ORTS. (2015). Obtenido de la web http://skatgame.net/mburo/orts/

- [PCGWiki, 2015] Procedural Content Generation Wiki. (2015). *Algorithms for Procedural Content Generation.* Obtenido de la web http://pcg.wikidot.com/category-pcg-algorithms

- [Picard, 1995] Rosalind W Picard. (1995). *Afective computing.* MIT Media Laboratory.

- [PWC, 2015] PWC. (2015). Obtenido de la web http://www.pwc.com/gx/en/industries/entertainment-media/outlook/segment-insights/video-games.html

- [Quantic, 2015a] Quantic Dream. (2015). *Fahrenheit.* Obtenido de la web https://www.fahrenheit-game.com/

- [Quantic, 2015b] Quantic Dream. (2015). *Heavy Rain.* Obtenido de la web http://www.quanticdream.com/en/#!/en/category/heavy-rain

- [Rockstar, 2015] Rockstar Games. (2015). *Grand Theft Auto V.* Obtenido de la web http://www.rockstargames.com/V/es

- [Shaker et al., 2011] Shaker, N., Togelius, J., Yannakakis, G. N., Shimizu, T., Hashiyama, T., Sorenson, N., ... Baumgarten, R. (2011). The 2010 Mario AI Championship: Level Generation Track. *IEEE Transactions on Computational Intelligence and AI in Games*, 332-347.

- [Smith, 2010] Smith, A. M. & Mateas, M. (2010). Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games. *IEEE Conference on Computational Intelligence and Games,* 273-280.

- [Togelius et al., 2011] Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2011). Search-Based Procedural Content Generation: A Taxonomy and Survey*. IEEE Transactions on Computational Intelligence and AI in Games, 3*(3), 172-186.

- [Togelius et al., 2013] Togelius, J., Champandard, A. J., Lanzi, P. L., Mateas, M., Paiva, A., Preuss & M., Stanley, K. O. (2013). Procedural Content Generation: Goals, Challenges and Actionable Steps. *Artificial and Computational Intelligence in Games,* 61-75.

Referencias

- [Togelius et al., 2015] Togelius, J., Shaker, N., & Nelson, M. J. (2015). Introduction. En J. Togelius, N. Shaker, & M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer.

- [West, 2008] West, M. (2008). *Random Scattering: Creating Realistic Landscapes.* Obtenido de la web http://www.gamasutra.com/view/feature/130071/random_scattering_creating_.php

Anexo 1

Fichero XML de reglas

```
<Rules>
  <Missions>
    <Conquest>
      <info>Acción de enviar tropas a un planeta que no tiene dueño, es decir, no es propiedad de ningún otro jugador.</info>
      <unlimited>false</unlimited>
      <maxBuilders>100</maxBuilders>
      <maxTanks>100</maxTanks>
      <maxShips>100</maxShips>
      <maxPlanes>100</maxPlanes>
      <maxSoldiers>100</maxSoldiers>
      <maxWorkers>100</maxWorkers>
    </Conquest>
    <Invasion>
      <info>Acción de enviar tropas a un planeta que, actualmente, es propiedad de otro jugador.</info>
      <unlimited>false</unlimited>
      <maxBuilders>100</maxBuilders>
      <maxTanks>100</maxTanks>
      <maxShips>100</maxShips>
      <maxPlanes>100</maxPlanes>
      <maxSoldiers>100</maxSoldiers>
      <maxWorkers>100</maxWorkers>
    </Invasion>
    <Expansion>
      <info>Acción de enviar tropas a un planeta aliado, es decir, el planeta es propiedad del jugador que envía la misión.</info>
      <unlimited>false</unlimited>
      <maxBuilders>100</maxBuilders>
      <maxTanks>100</maxTanks>
      <maxShips>100</maxShips>
      <maxPlanes>100</maxPlanes>
      <maxSoldiers>100</maxSoldiers>
      <maxWorkers>100</maxWorkers>
    </Expansion>
  </Missions>
  <BattleMode>battle1</BattleMode>
  <Resources>
    <hungry>10</hungry>
    <libid>10</libid>
    <workerStr>30</workerStr>
    <soldierStr>50</soldierStr>
    <builderStr>30</builderStr>
    <airCombatStr>50</airCombatStr>
    <aquaticStr>50</aquaticStr>
    <terrestrialCombatStr>60</terrestrialCombatStr>
  </Resources>
  <EvolutionOptions>
    <naturalEvolution>true</naturalEvolution>
    <naturalFactor>1.0E-4</naturalFactor>
    <livingEvolution>true</livingEvolution>
    <livingFactor>1.0E-4</livingFactor>
    <beastFactor>1.0E-4</beastFactor>
    <recoveryFactor>1.0E-4</recoveryFactor>
    <industrialEvolution>true</industrialEvolution>
    <industrialFactor>1.0E-4</industrialFactor>
    <flightFatigue>true</flightFatigue>
    <fatigueFactor>1.0E-4</fatigueFactor>
  </EvolutionOptions>
</Rules>
```

Anexo 1: Fichero XML de reglas

Anexo 2

Fichero JSON de reglas

```
{
  "Missions" : {
    "Conquest" : {
      "info" : "Acción de enviar tropas a un planeta que no tiene dueño, es decir, no es propiedad de ningún otro jugador",
      "unlimited" : "false",
      "maxBuilders" : "100",
      "maxTanks" : "100",
      "maxShips" : "100",
      "maxPlanes" : "100",
      "maxSoldiers" : "100",
      "maxWorkers" : "100"
    },
    "Invasion" : {
      "info" : "Acción de enviar tropas a un planeta que, actualmente, es propiedad de otro jugador",
      "unlimited" : "false",
      "maxBuilders" : "100",
      "maxTanks" : "100",
      "maxShips" : "100",
      "maxPlanes" : "100",
      "maxSoldiers" : "100",
      "maxWorkers" : "100"
    },
    "Expansion" : {
      "info" : "Acción de enviar tropas a un planeta aliado, es decir, el planeta es propiedad del jugador que envía la misión",
      "unlimited" : "false",
      "maxBuilders" : "100",
      "maxTanks" : "100",
      "maxShips" : "100",
      "maxPlanes" : "100",
      "maxSoldiers" : "100",
      "maxWorkers" : "100"
    }
  },
  "BattleMode" : "battle1",
  "Resources" : {
    "hungry" : "10",
    "libid" : "10",
    "workerStr" : "30",
    "soldierStr" : "50",
    "builderStr" : "30",
    "airCombatStr" : "50",
    "aquaticStr" : "50",
    "terrestrialCombatStr" : "60"
  },
  "EvolutionOptions" : {
    "naturalEvolution" : "true",
    "naturalFactor" : "1.0E-4",
    "livingEvolution" : "true",
    "livingFactor" : "5.0E-4",
    "beastFactor" : "1.0E-4",
    "recoveryFactor" : "1.0E-4",
    "industrialEvolution" : "true",
    "industrialFactor" : "1.0E-4",
    "flightFatigue" : "true",
    "fatigueFactor" : "1.0E-4"
  }
}
```

Anexo 2: Fichero JSON de reglas

Anexo 3

Manual de usuario del editor de reglas

En este anexo se explicará el funcionamiento del editor de reglas implementado y cómo usarlo correctamente desde el punto de vista del usuario.

Estructura del editor

El editor está dividido en 4 sectores diferenciados (ver Figura 1): el árbol de reglas en la zona izquierda (marcado en color verde), la zona de edición de reglas a la derecha (marcada en color rojo), el panel de mensajes del sistema en la zona inferior (marcado en color azul) y la barra menú en la parte superior (marcada en color amarillo).

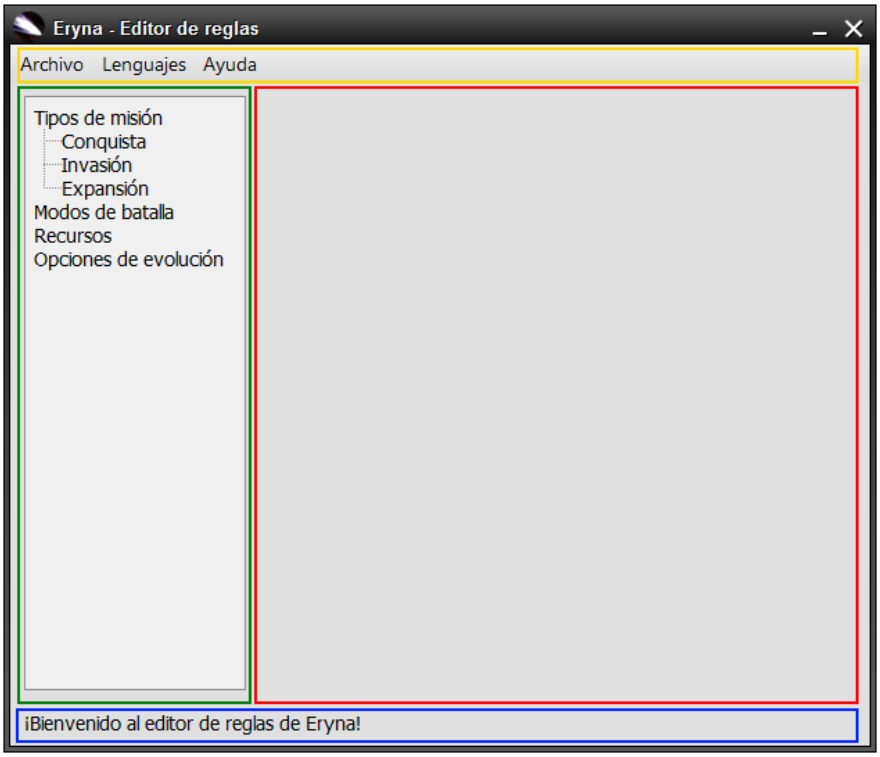

Figura 1. Pantalla principal del editor de reglas

Árbol de reglas y edición

En función del tipo de regla que se seleccione en el árbol de reglas aparecerá una pantalla u otra en la zona de edición de reglas:

Tipos de misión

Si se selecciona esta opción aparecerá la pantalla de edición de tipos de misión (ver Figura 2). Esta pantalla permite crear nuevos tipos de misión, editarlas y

eliminarlas. Las misiones originales del editor (Conquista, Invasión y Expansión) no pueden ser editadas ni eliminadas.

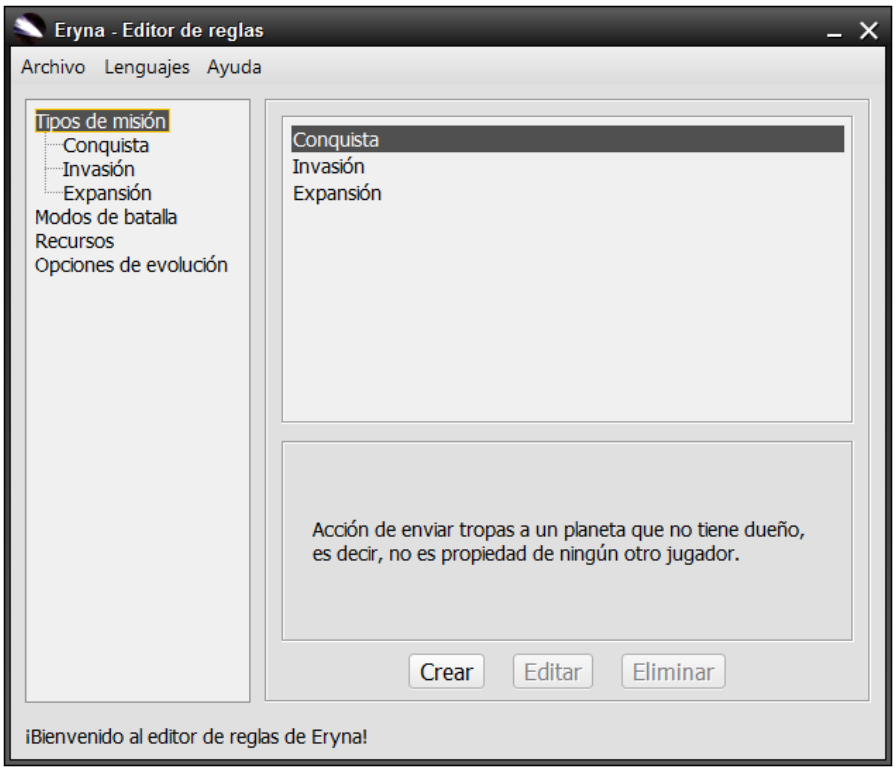

Figura 2. Pantalla de edición de misiones

Al seleccionar una misión de la lista de las disponibles para editar, aparece en el cuadro inferior su descripción. Si se desea crear una nueva misión, basta con usar el botón Crear y aparecerá una nueva pantalla destinada a ello (ver Figura 3, imagen de la izquierda). Igualmente, si se usa la opción de Editar sobre un tipo de misión que se haya creado, aparecerá una ventana muy parecida a la de creación (ver Figura 3, imagen de la derecha).

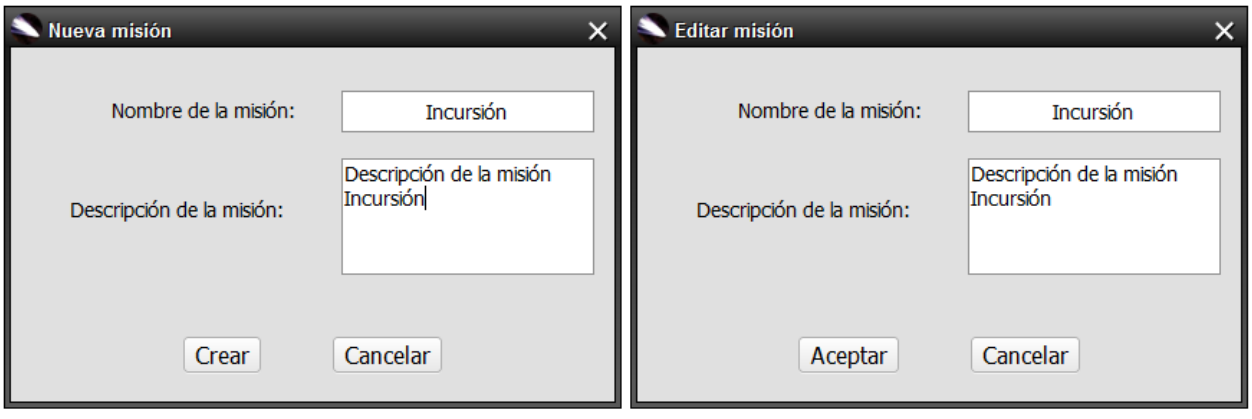

Figura 3. Pantallas de creación y edición de misiones

Por último, si se usa la opción de Eliminar una misión que se haya creado, aparecerá una ventana de advertencia para confirmar la eliminación o cancelarla (ver Figura 4).

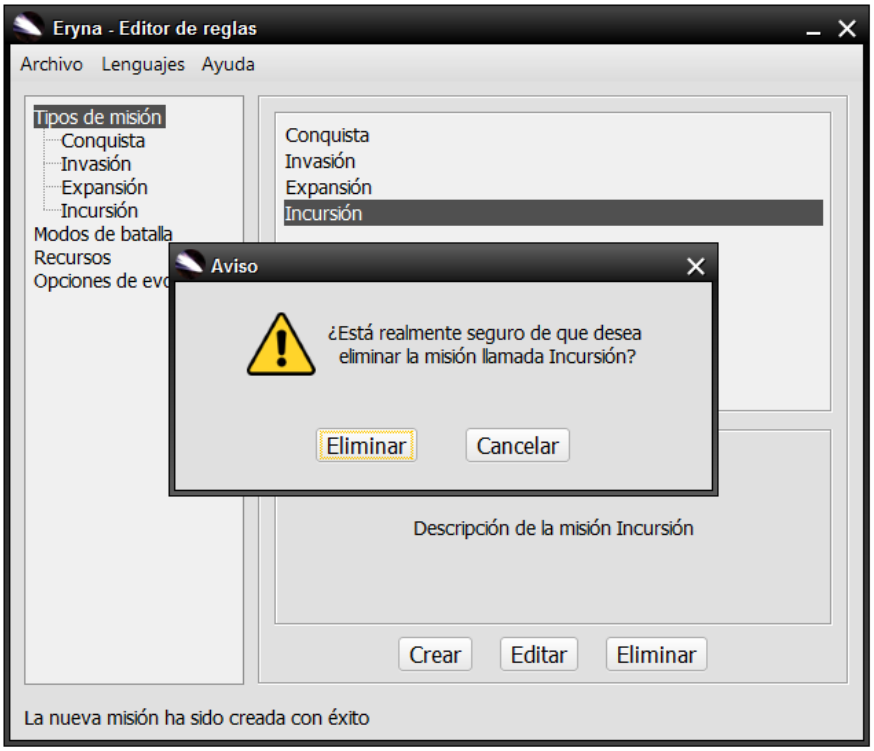

Figura 4. Mensaje de advertencia para eliminar misión

Misión: Conquista, Invasión, Expansión, etc.

Al seleccionar alguna misión en el árbol de reglas, se pueden editar sus restricciones: valores máximos para el envío de las diferentes unidades. En la Figura 5 aparece esta pantalla de edición de restricciones de la misión Invasión a modo de ejemplo. Todas las unidades pueden tener un valor máximo comprendido entre 0 y 100000, ambos inclusive.

Esta pantalla y algunas otras que se detallarán posteriormente tienen un botón de bloqueo. Esto permite bloquear y desbloquear cambios en las diversas pantallas de edición de reglas de forma que, si no se han introducido valores erróneos, al pulsar el botón de Bloquear la pantalla se bloqueará aceptando los cambios en las reglas. Si hubiese algún error, se mostrarán por pantalla (al final de este manual se repasan los mensajes de error que pueden aparecer). A la hora de guardar las reglas editadas en un fichero XML o JSON, todas deben estar bloqueadas correctamente.

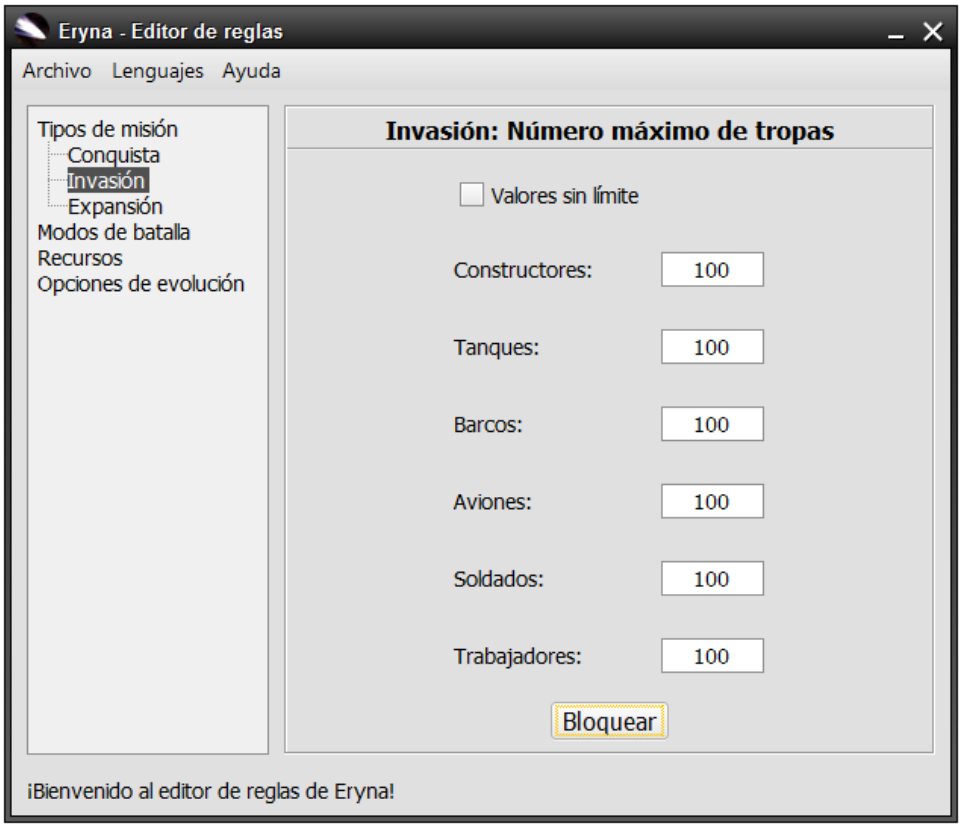

Figura 5. Pantalla de edición de restricciones de una misión

Modos de batalla

En esta pantalla se puede seleccionar el modo de batalla que el juego usará en caso de que las misiones de varios jugadores lleguen al mismo tiempo a un planeta. Cada una de las misiones predefinidas que aparecen aquí dispone de una descripción de su funcionamiento que aparece en el cuadro inferior al seleccionar cada una de ellas (ver Figura 6).

Recursos

En el caso de los recursos, se pueden editar diferentes valores para las reglas relativas a los recursos bióticos e industriales (ver Figura 7). Estos valores están expresados en porcentajes, por lo que sólo se permiten números entre 0 y 100, ambos inclusive.

Opciones de evolución

Si se selecciona opciones de evolución en el árbol de reglas, aparecerá la pantalla que permite editar las reglas de evolución para los recursos naturales, industriales y para los seres vivos (ver Figura 8). Se permite activar y desactivar la evolución de los diferentes elementos y, si se activan, se pueden editar los factores que influyen en dicha evolución.

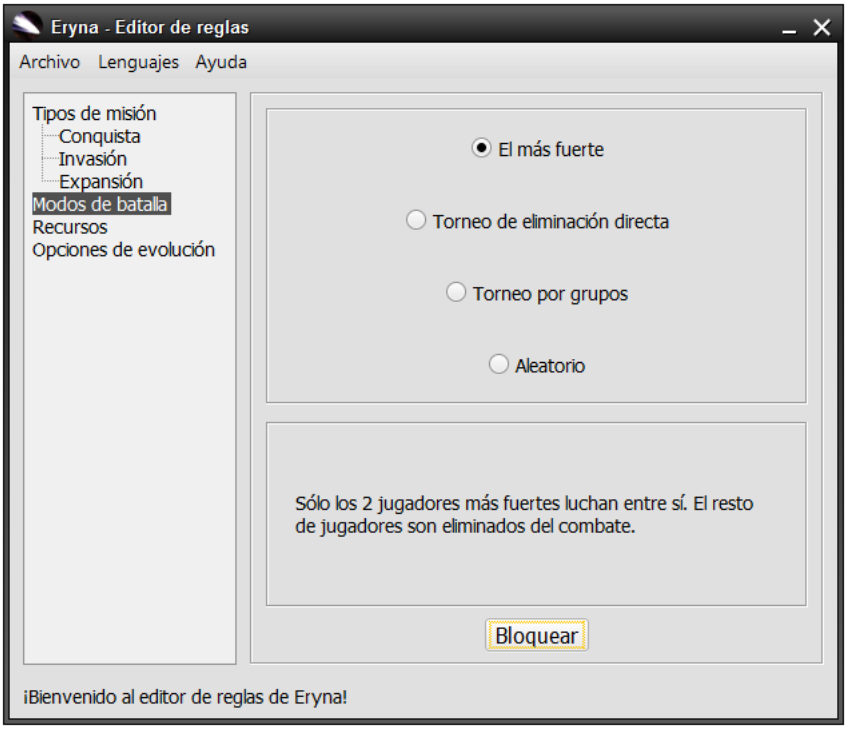

Figura 6. Pantalla de selección del modo de batalla

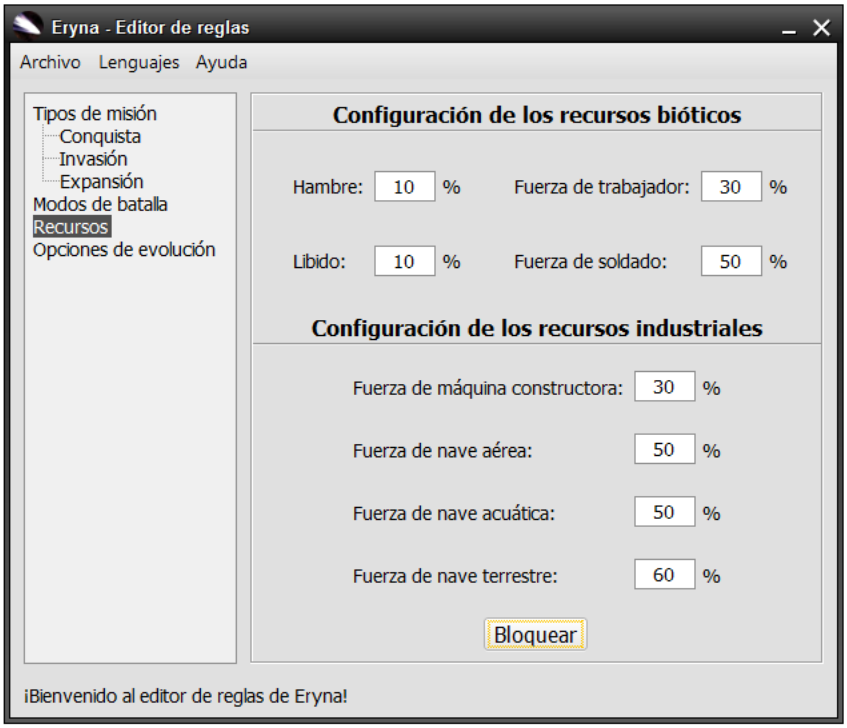

Figura 7. Pantalla de edición de recursos

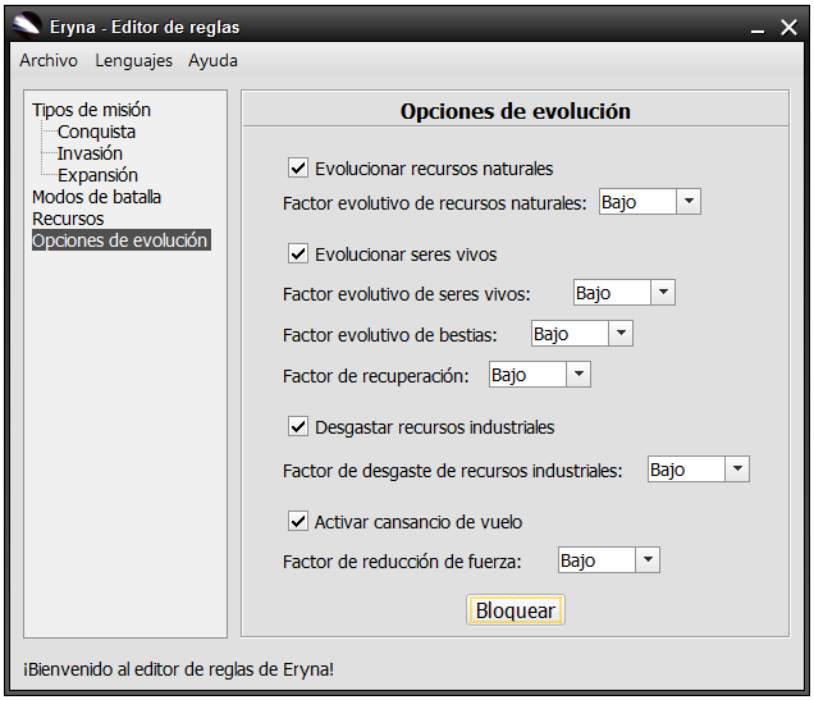

Figura 8. Pantalla de edición de opciones de evolución

Barra de menú

La barra de menú dispone de opciones tanto para guardar como cargar ficheros, cambiar el lenguaje del editor entre español e inglés y, además, mostrar una pantalla con ayuda para el usuario.

Cargar/Guardar archivo

Si se selecciona esta opción aparecerá una pantalla para explorar el ordenador y cargar o guardar el fichero deseado (ver Figura 9). Las opciones tanto para el cargado como el guardado son ficheros XML y JSON únicamente.

El atajo de teclado asociado a la carga de ficheros es *Ctrl* + *U* y el de guardado de ficheros es *Ctrl* + *S*.

Lenguajes

Bajo esta opción aparecen los idiomas Español e Inglés para seleccionar. En cualquier momento el usuario puede seleccionar una de estas opciones para cambiar de idioma el editor de reglas (ver Figura 10). La opción marcada por defecto en el editor es Español.

Ayuda

Bajo este botón aparece la opción de Ayuda de uso, la cual muestra una nueva pantalla independiente del editor con texto de ayuda al usuario explicando para qué sirve cada regla disponible en el editor en caso de duda sobre su funcionamiento (ver Figura 11). El atajo de teclado asociado a esta ayuda de uso es la tecla F1.

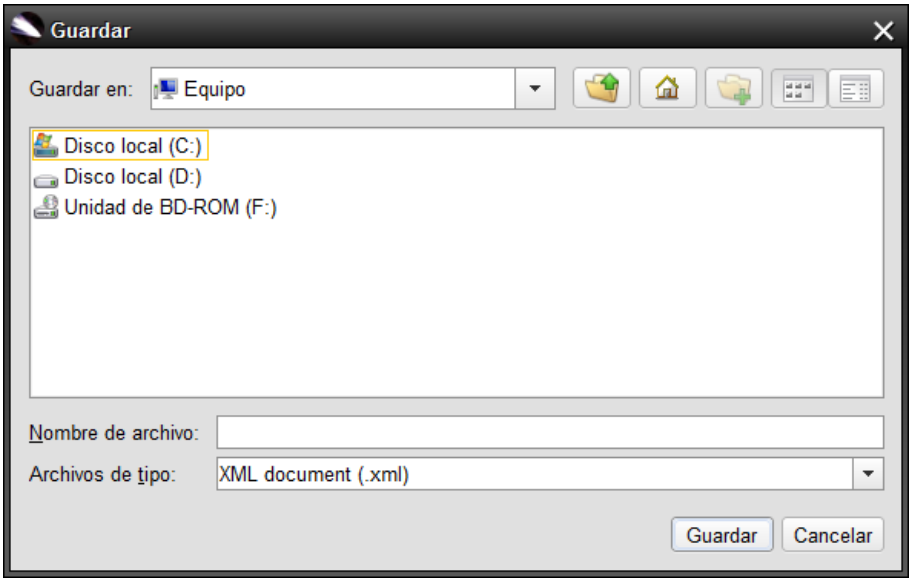

Figura 9. Pantalla de guardado de ficheros.

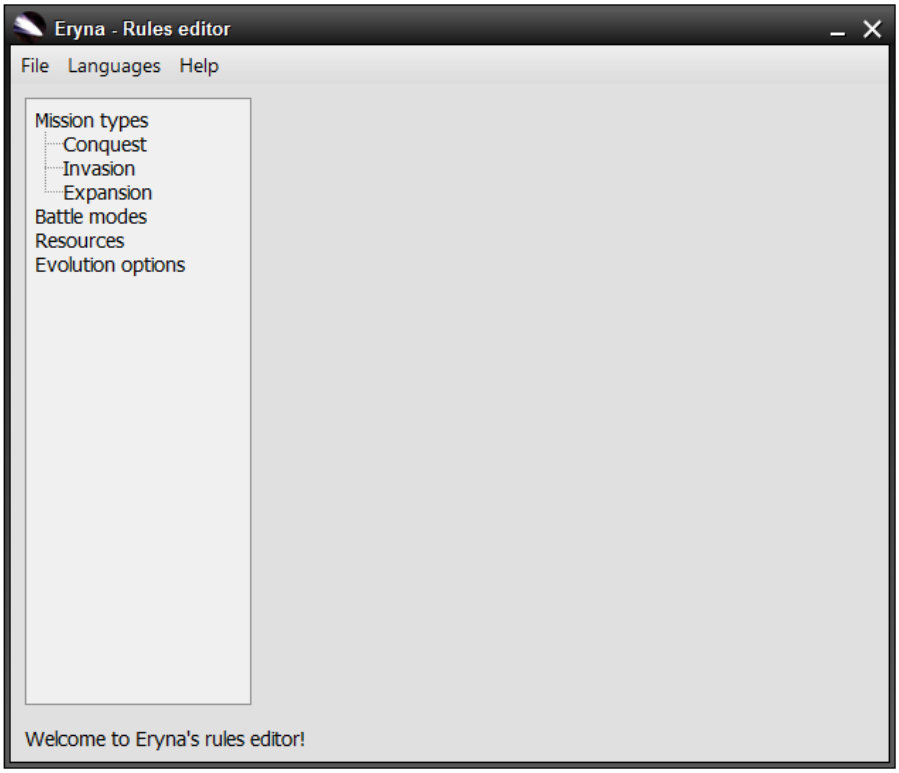

Figura 10. Pantalla principal en inglés.

Anexo 3: Manual de usuario del editor de reglas

Figura 11. Pantalla de ayuda al usuario.